Programmation Système Avancée - projet

1 Modalités

Le projet doit être réalisé en binôme (éventuellement, en monôme). Les soutenances auront lieu en mai, la date exacte sera communiquée ultérieurement. Pendant la soutenance, les membres d'un binôme devront chacun montrer leur maîtrise de la totalité du code.

Chaque équipe doit créer un dépôt git privé sur le gitlab de l'UFR :

`https://gaufre.informatique.univ-paris-diderot.fr`

dès le début de la phase de codage et y donner accès en tant que *Reporter* à tous les enseignants de cours et TP de Systèmes avancés.

Le dépôt devra contenir un fichier « `AUTHORS.md` » donnant la liste des membres de l'équipe (nom, prénom, numéro étudiant et pseudo(s) sur le gitlab). Vous êtes censés utiliser le gitlab de manière régulière pour votre développement. Le dépôt doit être créé **le 17 avril au plus tard**. Au moment de la création du dépôt, vous devez envoyer un mail à `zielonka@irif.fr` donnant la composition de votre équipe, et votre mail doit obligatoirement comme objet « `[syst av] projet` » (c'est important si vous ne voulez pas que votre mail se perde).

Le guide de connexion externe et la présentation du réseau de l'UFR se trouvent sur :
`http://www.informatique.univ-paris-diderot.fr/wiki/howto_connect`
et `http://www.informatique.univ-paris-diderot.fr/wiki/linux`

Le projet doit être accompagné d'un `Makefile` utilisable. Les fichiers doivent être compilés avec les options `-Wall -g` sans donner lieu à aucun avertissement (ni erreur bien évidemment) ; `make clean` devra supprimer tous les fichiers exécutables et les fichiers `*.o` de telle sorte que le `make` suivant permette de recompiler complètement le projet.

Si les conditions le permettent, la soutenance se fera à partir du code déposé sur le gitlab et **sur les machines de l'UFR** (salles 2031 et 2032) : au début de la soutenance, vous aurez à cloner votre projet à partir du gitlab et le compiler avec `make`. Préparez-vous pour que cela ne prenne pas 15 minutes : normalement ces deux tâches ne doivent pas prendre plus d'une minute, si ce n'est pas le cas alors vous aurez moins de temps pour présenter votre travail.

Vous devez fournir un jeu de tests permettant de vérifier que vos fonctions sont capables d'accomplir les tâches demandées, en particulier quand plusieurs processus lancés en parallèle demandent au serveur d'exécuter des fonctions.

Si vous avez des questions, merci de les poser dans le salon `projet` sur discord.

2 Local Procedure Call - LPC

Remote Procedure Call (RPC) est une interface qui permet d'appeler une fonction sur une machine distante.

Inspiré par RPC, le projet demande d'implémenter une interface *Local Procedure Call* (LPC) qui, intuitivement, permet à un processus client d'appeler une fonction implémentée dans un autre processus, le processus serveur.

Évidemment cela pose un problème de transmission : comment passer les paramètres, et comment récupérer les résultats d'un tel appel.

Les processus utiliseront pour cela la mémoire partagée obtenue par l'image mémoire d'un shared memory object (ShMO), qui sera créé par le serveur.

3 Fonctions à implémenter pour le client

Dans ce qui suit, nous allons utiliser les définitions suivantes :

```
typedef enum {STRING, DOUBLE, INT, NOP}  lpc_type;

typedef struct {
    int slen;
    char string[];
} lpc_string;
```

3.1 lpc_open

```
void *lpc_open( const char *name )
```

La fonction ouvre le shared memory object dont le nom est `name`, fait sa projection en mémoire et retourne l'adresse de cette mémoire. Si le shared mémory object n'existe pas, alors `lpc_open` échoue et retourne `NULL`.

Cette fonction sera utilisée par un processus client souhaitant appeler des fonctions du serveur.

3.2 lpc_close

```
int lpc_close( void *mem )
```

C'est la fonction à exécuter par un client lorsqu'il ne veut plus faire d'appels aux fonctions du serveur. Essentiellement cette fonction fera juste un appel à `munmap` (et éventuellement d'autres nettoyages comme la suppression de structures que le client aurait pu allouer localement).

3.3 lpc_call

```
int lpc_call( void *memory, const char *fun_name, ...)
```

Un processus client utilisera `lpc_call` pour faire appel à une fonction du serveur ; `lpc_call` est une fonction à nombre variable de paramètres [1] :

- `memory` est l'adresse de la mémoire partagée retournée par `lpc_open` ;
- `fun_name` est le nom de la fonction appelée ;
- les paramètres suivants sont les paramètres de la fonctino appelée.

Pour simplifier, on suppose que les seuls paramètres utilisables sont de type pointeur : `int *`, `double *`, `lpc_string *` (le type `lpc_string` a été défini au début de la section 3).

Pour parcourir ses arguments, une fonction à nombre variable de paramètres doit « connaître » le type de chaque paramètre. Pour cette raison les paramètres variables de `lpc_call` viennent par deux : dans chaque couple, le premier paramètre sera de type `lc_type` et le deuxième un des trois types pointeurs, comme ci-dessous :

- `INT`, `int *`
- `DOUBLE`, `double *`
- `STRING`, `lc_string *`

Par exemple, après un paramètre `lpc_type` avec la valeur `DOUBLE`, vient un paramètre de type `double *` qui donne l'adresse d'une donnée de type `double`.

La liste de paramètres terminera par le marqueur de fin `NOP` (de type `lc_type`).

3.3.1 STRING

La structure `lpc_string` sert à transmettre des chaînes de caractères au serveur, et à récupérer les chaînes de caractères construites par le serveur (on ne fait pas de distinction entre les paramètres d'entrée et de sortie, ou plus exactement chaque paramètre peut servir comme paramètre d'entrée et comme paramètre de sortie).

Rappelons que tous les tableaux qui font partie d'une structure doivent être de taille fixe, sauf le tableau qui occupe le dernier champ de la structure.

Dans la définition

```
typedef struct {
    int slen;
    char string[];
} lpc_string;
```

`string` à la fin est de taille 0, ce qui n'a aucune utilité si on utilise `lpc_string` directement, c'est-à-dire en déclarant une variable

```
lpc_string x;
```

La variable `x` ainsi déclarée possède juste la mémoire suffisante pour le champ `slen`, mais pas pour le tableau `string`.

1. Vous pouvez par exemple consulter la section 4.9 de `https://www.rocq.inria.fr/secret/Anne.Canteaut/COURS_C/cours.pdf` si vous avez besoin de rafraîchir vos connaissances des fonctions à nombre variable de paramètres en C.

En revanche, quand la structure `lpc_string` est allouée dynamiquement, on peut réserver la mémoire pour `string`. Par exemple pour passer comme paramètre la chaîne de caractères `"bonjour"`, on peut procéder de façon suivante :

```
char *a="bonjour";
size_t  lo = strlen(a) + 1; /* +1 pour le char nul qui termine la chaîne
                             * de caractères*/
lpc_string *s = malloc( sizeof( lpc_string ) + lo + 1);
if( s == NULL }{
  /* gérer l'erreur de malloc */
}

s->slen = strlen(a) + 1;  /* mettre dans le champ slen la longueur de
                           * tableau string */

strcpy( s->string, a );   /* copier a dans le tableau string */

/* maintenant s peut être utilisé comme paramètre de lpc_call. */
```

En général le champ `slen` de la structure `lpc_string` sert à stocker la taille du tableau `string`.

Pour faciliter la construction des objets `STRING` vous devez écrire la fonction :

```
lpc_string *lpc_make_string( const char *s, int taille )
```

qui implémente le comportement suivant :

- si `taille > 0` est `s == NULL`, `lpc_make_string` alloue la mémoire pour `lpc_string` avec le champ `string` de taille `taille` ; `slen` prendra la valeur `taille`. On initialise le tableau `string` avec le caractère nul (utiliser `memset`) ;
- si `taille <= 0` et `s != NULL`, alors `lpc_make_string` alloue la mémoire avec le tableau `string` de `strlen(s)+1` octets et y copie la chaîne de caractères pointée par `s` ; `taille` prendra la valeur `strlen(s)+1` ;
- si `taille >= strlens + 1`, la fonction `lpc_make_string` alloue la structure avec le champ `string` de `taille` octets et y copie `s`. Le champ `slen` prendra la valeur `taille` ;
- dans tous les autres cas `lpc_make_string` ne fait pas d'allocation mémoire et retourne `NULL`.

3.4 Exemple

Dans cet exemple, on suppose que le client veut appeler une fonction `fun_difficile` du serveur qui prend en entrée un `int`, un `double` et un `lpc_string` et peut produire en sortie aussi bien un `int`, qu'un `double` ou un `lpc_string`.

On supposera toujours que chaque paramètre entrant est aussi potentiellement un paramètre sortant ; c'est pour cette raison qu'on utilise les pointeurs qui permettent de modifier les données dans la fonction.

On procédera de la façon suivante pour préparer l'appel à `lpc_call` :

```
int a = 15;
double b = 0;

/* on alloue un objet lpc_string avec un tableau string de 100 octets. C'est
 * beaucoup plus que ce qu'il faut pour stocker "bonjour". En prévoit que
 * fun_difficile remplacera "bonjour" par un autre string, bien plus long,
 * jusqu'à 99 caractères. On alloue assez de mémoire pour ce nouveau string. */
lpc_string *s = lpc_make_string("bonjour", 100);


int r = lpc_call( mem,  "fun_difficile", INT, &a, DOUBLE, &b, STRING, s, NOP);
```

Le paramètre `mem` est l'adresse de la mémoire partagée par le client et le serveur, et le deuxième paramètre, `"fun_difficile"`, est le nom de la fonction à appeler.

`lpc_call` parcourt les paramètres variables.

En lisant la valeur `INT` du premier paramètre variable `lpc_call` « apprend » que le paramètre suivant est de type `int *` et copie la valeur de type `int` stockée à l'adresse passée en paramètre (autrement dit, la valeur de `a`) vers la mémoire partagée.

La valeur du paramètre suivant est `DOUBLE`. Donc `lpc_call` sait que le paramètre suivant est de type `double *` et `lpc_call` copie la valeur de type `double` qui se trouve à l'adresse passée en paramètre (autrement dit, la valeur de `b`) vers la mémoire partagée.

Ensuite, `lpc_call` tombe sur `STRING` donc `lpc_call` déduit que le paramètre suivant est de type `lpc_string *`. Le nombre d'octets à copier dans la mémoire partagée dépend de la taille du tableau `string` dans la structure `lpc_string`, qui se trouve dans le champ `slen` de la même structure. Donc `lpc_call` pourra facilement calculer le nombre d'octets à copier.

Et finalement, on a la valeur `NOP` qui indique qu'il n'y a plus de paramètres.

En résumé, les valeurs `INT`, `DOUBLE`, `STRING` servent à indiquer le type de pointeur qui suit, et `lpc_call` commence par copier dans la mémoire partagée les données qui se trouvent aux adresses indiquées par les pointeurs.

La disposition de la mémoire partagée sera décrite plus en détail dans la section 4.

Après avoir recopié les valeurs d'entrée dans la mémoire partagée, on a la suite d'actions ci-dessous :

(a) `lpc_call` « réveille » le serveur (sans doute à l'aide de `pthread_cond_signal`) et se suspend par un appel à `pthread_cond_wait` ;

(b) le serveur appelle la fonction appropriée ;

(c) quand cette fonction termine, le serveur recopie les nouvelles valeurs des paramètres (dans notre exemple, les nouvelles valeurs `int`, `double` et `lpc_string`) dans la mémoire partagée et il « réveille » le client par un appel à `pthread_cond_signal` ;

(d) le client recopie les nouvelles valeurs depuis la mémoire partagée vers les adresses données en paramètres de `lpc_call` ;

(e) l'appel à `lpc_call` termine.

Pour résumer, le client est responsable du transfert de données (paramètres) de sa mémoire vers la mémoire partagée, et ensuite du transfert des valeurs calculées depuis la mémoire partagée vers sa mémoire.

Le serveur est responsable du transfert de données depuis la mémoire partagée vers sa propre mémoire, il doit identifier la fonction que le client veut appeler, il doit appeler cette fonction, et à la fin de l'appel il doit transférer les résultats vers la mémoire partagée.

Notez que l'appel à `lpc_call` ne doit pas retourner tant que le transfert de résultats vers la mémoire client n'est pas terminé. Du point de vue du client, l'appel à `lpc_call` a l'air d'un appel de fonction normal : lors de l'appel, les variables `a,b,c` dont les adresses sont passées en paramètres de `lpc_call` contiennent des valeurs d'entrée, et quand l'appel à `lpc_call` termine, le client retrouve dans ces variables `a, b, c` de nouvelles valeurs calculées par la fonction `fun_difficile`.

4 Mémoire partagée

La mémoire partagée peut être divisée en deux parties.

La première partie est une structure `header` (à vous de la définir en détail) qui contient toutes les informations nécessaires pour implémenter un *Local Procedure Call* (les mutex, les conditions pour synchroniser l'accès mémoire de clients et de serveur). Dans cette partie, nous pouvons aussi réserver la mémoire pour la valeur `int` retournée par la fonction appelée et la mémoire pour la valeur de `errno`, voir la section 6.

La deuxième partie de la mémoire partagée, appelons-la `DATA`, est la zone où `lpc_call` met les données pour la fonction appelée : c'est ici que `lpc_call` a copié les `int`, `double` et `lpc_string` de l'exemple de la section précédente.

5 Le serveur

Le serveur doit maintenir un tableau de structures :

```
#define NAMELEN 48
typedef struct{
    char fun_name[NAMELEN];
    int (*fun)(void *);
} lpc_function;
```

Chaque élément du tableau correspond à une fonction que le serveur peut exécuter pour un client.

Le champ `fun_name` donne le nom de la fonction. Le champ `fun` est un pointeur de fonction qui contient l'adresse de la fonction.

Comme le montre la définition du champ `fun`, toutes les fonctions auront la même signature : un seul paramètre d'entrée de type `void *` et un `int` comme valeur de sortie. Rappelons que chaque fonction peut retourner deux valeurs : 0 quand l'appel réussit et -1 sinon.

Toutes les données de la fonction appelée se trouvent regroupées à l'adresse passée comme paramètre de la fonction `fun`. Le format de ces données est exactement le même que dans la partie `DATA` de la mémoire partagée (voir la section 4).

En fait on peut même imaginer que, pour exécuter la fonction `fun` pour le client, le serveur passe l'adresse de la partie `DATA` de la mémoire partagée comme l'argument de `fun`. Cela éviterait la copie de données de la mémoire partagée vers la mémoire privée du serveur.

6 Gestion d'erreurs

Comme le but de `lpc_call` est de simuler le plus fidèlement possible un appel de fonction normal, il faut aussi simuler le comportement d'un appel de fonction qui provoque une erreur.

Supposons par exemple que le serveur exécute pour un client une fonction `fun` qui utilise l'appel système `open`, et que cet appel échoue. La fonction `fun` retournera -1 et dans ce cas le serveur doit copier la valeur de `errno` dans la mémoire partagée, et ensuite `lpc_call` doit copier cette valeur depuis la mémoire partagée dans la variable `errno` du processus client.

6.1 Mémoire insuffisante dans `lpc_string`

Il reste la question de données de type `lpc_string`. La taille du tableau `string` dans cette structure est fixée par le client. Mais si l'objet `lpc_string` est utilisé pour la sortie, la fonction appelée peut trouver qu'il n'y a pas d'assez de mémoire dans le tableau `string` de `lpc_string` pour y mettre la chaîne de caractères obtenue comme résultat.

Dans ce cas, la fonction appelée doit :

- mettre la valeur `ENOMEM` dans `errno` [2] ;
- mettre la valeur -1 dans le champ `slen` de `lpc_string` correspondant (si la fonction appelée utilise plusieurs données de type `lpc_string`, il faut que le client puisse identifier quel est le`lpc_string` qui pose le problème d'insuffisance de mémoire) ;
- retourner -1.

7 Fonctions du serveur

C'est à vous de voir quelles sont les fonctions nécessaires pour implémenter le serveur.

L'utilité de certaines fonctions est évidente, par exemple une fonction qui, en utilisant un nom de fonction, cherche le pointeur de fonction correspondant dans le tableau décrit au début de la section 5.

Une autre fonction nécessaire :

```
1 void *lpc_create( const char *nom, size_t capacite )
```

2. Le code d'erreur qui signale qu'il n'y a pas assez de mémoire

La fonction `lpc_create()` sera appelée par le serveur. Elle crée un ShMO de la taille de `capacite * taille_de_page` octets. (Si l'objet ShMO existe alors `lpc_create` modifiera sa taille pour que l'objet ait `capacite * taille_de_page` d'octets.)

Ensuite `lpc_create` fait la projection de l'objet ShMO en mémoire et retourne l'adresse de cette mémoire. La fonction initialise le contenu de la partie `header` de la mémoire partagée (initialisation de mutexes et de conditions).

`lpc_create` retournera le pointeur vers la mémoire partagée, ou `NULL` en cas d'échec.

8 Serveur distribué

Le projet tel qu'il est présenté ci-dessus propose une implementation simple de lpc. Cette implementation a un défaut majeur : à un moment donné un seul appel de fonction client est effectivement exécuté par le serveur, c'est-à-dire à chaque moment un seul processus client est servi par le serveur. Si l'appel de fonction client est long tous les autres clients doivent attendre bloqués sur `lpc_call`.

Le serveur simple présenté dans les sections précédentes permet de valider le projet mais pour une très bien il faut développer un serveur distribué qui peut servir plusieurs appels de fonctions en même temps.

La suite de cette section décrit le comportement du serveur distribué.

Quand le serveur reçoit un appel lpc il créa un processus enfant. C'est cet enfant qui se chargera de l'exécution de la fonction client et de toute communication avec le client. Pendant ce temps le serveur peut répondre à d'autres appels fonction de la part d'autres clients.

Il reste un problème à résoudre, les différents clients ne peuvent pas mettre les données dans la même mémoire ou plus exactement dans le même emplacement dans la mémoire. Une solution consisterait à subdiviser la partie DATA de la mémoire partagée en segments, chaque segment correspond à un appel fonction. C'est difficile à gérer. On propose ici une autre solution.

Soit `shmo_name` le nom de shared memory object qui est utilisé par l'implementation simple de local procedure call.

Quand le client exécute `lpc_call` avec un serveur distribué le client utilise la mémoire partagée liée à `shm_name` pour passer son `pid` au serveur.

L'enfant du serveur (celui qui doit gérer l'appel de client) créa un nouveau shared memory object dont le nom est obtenu en concatenant `shmo_name` avec le pid du client. Nous appelons ce nouveau shared memory object *ShMO de communication*.

Le processus client [3] crée aussi l'image mémoire du ShMO de communication. Tout passage de données entre le client et l'enfant du serveur se fera en utilisant cette mémoire partagée. Le format de cette mémoire partagé est exactement le même que pour le serveur simple. En fait à partir de ce moment la communication entre le processus client et l'enfant du

3. Quand je parle de processus client, je parle de la fonction `lpc_call`. Toute implementation de local procedure call du côté client est faite dans la fonction `lpc_call`, sauf l'initialisation dans `lpc_open` et fermeture dans `lpc_close`.

serveur est la même que la communication entre le serveur simple et le client, c'est-à-dire le client et le processus enfant du serveur interagissent comme dans le cas de serveur simple.

Une fois `lpc_call` transmet les résultats de local procedure call vers la mémoire du client, avant terminé `lpc_call` doit indiquer au processus l'enfant du serveur que le transfert est fini (un flag dans la mémoire partagée). L'enfant du serveur détruit l'objet shmo de communication et termine.

Le serveur devra de temps en temps supprimer les enfants zombies.

9 Organisation du code

Toutes les fonctions du client doivent être regroupées dans le fichier `lpc_client.c`. Les fonctions auxiliaires que le client n'utilise pas directement doivent êtres déclarées `static`. Les fonctions directement utilisées par le client et décrites dans la section 3 seront déclarées dans le fichier d'en-tête `lpc_client.h` correspondant. Tout programme qui utilise LPC utilisera une inclusion de ce dernier fichier.

Le code du serveur peut être divisé en deux parties très largement indépendantes.

D'un côté le serveur contient les fonctions qu'il appelle pour le client. Chacune de ces fonctions doit être définie dans un fichier séparé. Dans la suite nous appelons ces fonctions `fonctions client`. Le contenu des fonctions client n'a aucune importance pour le serveur. Les fonctions client font partie du serveur mais doivent être faciles à ajouter ou supprimer.

Le serveur contient aussi les fonctions nécessaires pour implémenter la logique du serveur, et qui assurent le bon fonctionnement du serveur indépendamment des fonctions client. Cette partie du serveur doit être séparée. Appelons cela la base du serveur. La fonction `main()` du serveur fait partie de la base. Du point de vue du serveur, peu importe quelle est la fonction appelée par le client, le serveur doit juste identifier le bon pointeur de fonction et assurer la circulation de données. La base doit changer le moins possible quand on ajoute ou on supprime une fonction client.

Pour tester LPC, il nous faut quelques fonctions client. Il n'est pas demandé que ces fonctions fassent quoi que ce soit d'intéressant. Elles servent uniquement pour tester le bon fonctionnement de LPC. Donc il faut des fonctions qui prennent plusieurs types de données `int`, `double`, `lpc_string` et qui modifient ces données. Pour tester la gestion d'erreurs par LPC, il faut des fonctions qui retournent -1 et qui mettent une valeur dans `errno`. Et finalement il faut des fonctions qui signalent que la taille de tableau `string` dans `lpc_string` est trop petite pour vérifier si le signalement d'erreur fonctionne bien dans ce cas.

Et finalement il faut un programme ou plusieurs programmes pour tester le bon fonctionnement de votre implémentation de LPC. Testez les appels qui réussissent pour voir si on récupère correctement les valeurs `int`, `double` ou `lpc_string`.

Testez aussi si votre LPC réagit correctement sur les erreurs de fonctions client.