Programmation des Systèmes

Manipulation de processus légers

Philippe MARQUET
et les équipes pédagogiques successives*

Ce document est le support de travaux dirigés et de travaux pratiques relatifs au processus légers. On utilisera particulièrement l'interface POSIX `pthread`.

1 Création de processus légers

Exercice 1 (Arbre de calcul)
Soit le programme suivant de calcul du n^e nombre de Fibonacci :

```
int
fib (int n)
{
    if (n<2)
        return n;
    else {
        int x, y;

        x = fib(n-1);
        y = fib(n-2);

        return x + y;
    }
}
```

```
int
main (int argc, char *argv[])
{
    int n, res;

    n = atoi(argv[1]);
    res = fib(n);

    printf("Fibo (%d) = %d\n", n, res);
    exit(EXIT_SUCCESS);
}
```

Question 1.1 On désire attacher un processus léger à chacune des instances de la fonction `fib()`. On construit ainsi un arbre de processus légers. Avant de retourner son résultat, un processus léger attend que ses deux fils aient terminé. □

Question 1.2 On désire maintenant changer le prototype de la fonction de calcul associée à un thread pour que le résultat ne soit plus retourné par la fonction, mais rangé à une adresse passée en paramètre. □

Exercice 2 (Recherche parallèle)
On désire réaliser une implantation multithreadée de la recherche d'une valeur `v` dans un tableau de valeurs selon la méthode suivante :

— on délègue la recherche de la valeur dans la première moitié du tableau à un thread ;
— on recherche la valeur dans la seconde moitié du tableau (en utilisant la même méthode de recherche parallèle) ;

Édition de liens avec la bibliothèque `pthread`

L'option `-lpthread` doit être ajoutée lors de l'édition de liens pour explicitement indiquer l'utilisation de la bibliothèque `pthread`.
On ajoutera en conséquence la ligne suivante dans son `Makefile` :

```
LDFLAGS = -lpthread
```

— on récupère le résultat de la recherche effectuée par le thread pour le combiner au résultat de notre recherche.

Soit la fonction

```
int search(int *tab, unsigned int size, int (*pred)(int));
```

de recherche d'une valeur entière satisfaisant la fonction prédicat `pred()` dans le tableau `tab` de `size` éléments. On cherche donc une valeur `v=tab[i]` telle que `pred(v)`.

Question 2.1 Donnez l'implantation d'une fonction `pt_search()` de prototype compatible avec la fonction principale d'un thread. Cette fonction `pt_search()` réalise la recherche par un simple appel à `search()`. □

Question 2.2 Donnez une implantation de `search()`. □

2 Exclusion mutuelle

Exercice 3 (Compteur)
Il s'agit de développer une version « MT-Safe » de la fonction suivante de gestion d'un compteur (par exemple un compteur du nombre d'appels...).

```
int
count_me(int i)
{
    static int count = 0; /* variable globale à visibilité locale */

    count += i;

    return count;
}
```

Question 3.1 Quel problème peut survenir lors de l'utilisation de cette fonction dans un programme multithreadé ? □

Question 3.2 Quelle structure utiliser pour protéger les accès à `count` ? □

Question 3.3 Comment retourner la valeur de `count` ? □

Exercice 4 (Retour sur l'arbre de calcul)
On propose d'améliorer la solution de l'exercice 1 de calcul du n^e nombre de Fibonacci de la manière suivante : plutôt que de créer un processus léger alors que la valeur a déjà été produite auparavant par un autre processus léger, on va récupérer cette valeur. Pour cela, on garde les résultats intermédiaires produits dans un tableau partagé entre les threads. (Les éléments de ce tableau sont par exemple initialisés à la valeur `INCONNUE` pour dénoter que la valeur n'a pas encore été calculée.)

```
#define INCONNUE       -1
static int fibtab [];
```

Squelette d'un serveur de chat multithreadé

Les fichiers sources d'un squelette de serveur de chat multithreadé sont accessibles sur le portail.
Vous y trouverez :

- un `Makefile` ;
- `mtcs.c`, point d'entrée du programme. Ouvre le port de connexion du serveur, écoute sur ce port et établit la connexion pour chaque nouveau client. Délègue à `manage_cnct()` la création d'un thread en charge de la gestion de cette connexion ;
- `cnct.c` et `cnct.h`, gestion de la connexion d'un client ;
- `tools.c` et `tools.h`, outils divers ;
- `config.h`, configuration du serveur.

La version fournie de la fonction `manage_cnct()` n'est pas multithreadée ; la connexion d'un client bloque le serveur...
Seul le fichier `cnct.c` est à modifier. Vous fournirez aussi des fichiers `stat.h` et `stat.c` pour la gestion des statistiques.
Il est possible d'activer la production de traces d'exécution du serveur par l'option `-v`. Utilisez les fonctions `pgrs*()` pour ce faire.

Proposez un mécanisme pour assurer le partage des accès à ce tableau `fibtab` entre tous les processus légers. Donnez la nouvelle implémentation de la fonction `fib()`. □

Exercice 5 (Serveur de chat)
Un serveur de chat multithreadé est basé sur le principe suivant :

- un thread maître assure l'écoute sur une socket de connexion ;
- à chaque nouvelle demande de connexion, il crée une socket pour interagir avec le client et délègue cette interaction à un thread par un appel à la fonction

```
int manage_cnct(int fd);
```

- cette fonction est en charge de créer un thread et d'enregistrer le descripteur `fd` comme celui d'un nouveau client ;
- ce thread devra lire en boucle sur la socket puis répéter sur l'ensemble des sockets des autres clients.

Dans une version initiale non multithreadée, l'implémentation de `manage_cnct()` repose sur un simple appel à `repeater()`.

Question 5.1 Donnez une implantation de la fonction `manage_cnct()` déléguant à un thread l'appel à la fonction.
On se préoccupera aussi de la terminaison de ce thread. □

Question 5.2 Quelle structure de données doit être partagée entre les threads pour réaliser cette répétition des flux d'entrée ? Identifiez les accès qui seront réalisés à cette structure de données. Comment les protéger ? □

On désire ajouter le maintien de statistiques au serveur de chat. Les informations suivantes seront mémorisées :

- nombre de clients ;
- nombre de lignes reçues ;
- nombre de lignes envoyées ;
- nombre maximal de client à un instant donné ;
- nombre maximal de lignes reçues d'un client ;
- nombre maximal de lignes envoyées à un client.

De plus, on désire pouvoir interroger le serveur sur ces statistiques en lui envoyant un signal `SIGUSR1`.

Utilisation du serveur de chat multithreadé

Au démarrage du serveur, le numéro du port d'écoute est affiché :

```
% ./mtcs
Server open on port 8012
...
```

il faut repérer ce numéro de port d'écoute (ici 8012), il faut aussi identifier aussi le nom de la machine ! (`localhost` pour la machine locale peut suffire).
On peut alors lancer des clients. Depuis un autre terminal :

```
% telnet localhost 8012
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
mtcs : bienvenu
fooooooooooooooo
fooooooooooooooo
FOOOOOOOOOOOOOOOoooooooooooooooooooooooooooooo.............
FOOOOOOOOOOOOOOOoooooooooooooooooooooooooooooo.............
```

on termine par `control-]` puis `quit` :

```
^]
telnet> quit
Connection closed.
```

Question 5.3 Comment garder et mettre à jour ces informations de statistiques ? □

Question 5.4 Pourquoi est-il dangereux de contrôler par un verrou mutex POSIX l'accès partagé aux informations de statistiques entre les threads produisant ces statistiques et le handler de signaux affichant ces statistiques ? □

Question 5.5 Implantez cette prise de statistiques et leur affichage. □

3 Synchronisation de processus légers

Exercice 6 (À la recherche du signal perdu...)
Soit l'extrait de code producteur/consommateur suivant :

```
/* Consommateur */                /* Producteur, Signaleur */

pthread_mutex_lock(&lock);        expression = TRUE;
while (!expression)               pthread_cond_signal(&cond);
  pthread_cond_wait(&cond, &lock);

do_thing();
pthread_mutex_unlock(&lock);
```

Question 6.1 Montrer que le consommateur peut ne pas être réveillé alors qu'il le devrait : des `pthread_cond_signal()` pouvant être perdus. □

Question 6.2 Quelle(s) correction(s) apporter à cet extrait de code pour éviter ce comportement ? □

Exercice 7 (Synchronisation de l'arbre de calcul)
On propose encore d'améliorer la solution des exercices 1 et 4 de calcul du n^e nombre de Fibonacci de la manière suivante : on modifie la dernière solution de mémorisation des calculs précédents pour marquer qu'un processus léger est en train de calculer une valeur. Les éléments du tableau `fibtab` sont soit une valeur positive, soit une des deux valeurs `INCONNUE` et `ENCOURS`.

```
#define INCONNUE       -1
#define ENCOURS        -2
static int fibtab [];
```

Identifiez la synchronisation nécessaire entre un processus léger en cours de calcul d'une valeur et un processus léger ayant besoin de cette valeur. Proposez une implémentation de cette synchronisation. □

Exercice 8 (Barrières de synchronisation)
On désire implanter une bibliothèque de barrières de synchronisation à l'aide des constructions fournies par la bibliothèque POSIX.

L'interface de cette bibliothèque sera la suivante :

```
typedef struct barrier_s barrier_t;

barrier_t *barrier_init(unsigned n_clients);
void barrier_destroy(barrier_t *barrier);
void barrier_wait(barrier_t *barrier);
```

La fonction `barrier_init()` alloue et initialise une barrière qui sera utilisée par `n_clients`.

La fonction `barrier_destroy()` demande la destruction de la barrière. Aucun thread ne doit attendre sur la barrière lors de la destruction.

La fonction `barrier_wait()` est appelée par un thread pour rentrer dans la barrière. Quand `n_clients` threads ont atteint la barrière, ils sont tous libérés.

Question 8.1 Quelle structure est nécessaire à l'implantation des barrières ? □

Question 8.2 Donner le code des fonctions de la bibliothèque. □

On considère maintenant l'utilisation suivante de la bibliothèque :

```
#define N      ...
barrier_t b;

static void *
foo(void *dummy)
{
    do_thing1();
    barrier_wait(&b);

    do_thing2();
    barrier_wait(&b);

    pthread_exit(0);
}
```

```
int
main(void)
{
    int i;
    pthread_t dummy_tid;
    b = barrier_init(N);

    for (i=1; i<N; i++)
        pthread_create(&dummy_tid, NULL,
                       foo, NULL);
    foo();
}
```

Question 8.3 Considérez les différents ordonnancements des threads pouvant intervenir entre la première et la seconde utilisation de la barrière. Mettez en évidence un fonctionnement erroné possible. □

Question 8.4 Corrigez la première implantation de la bibliothèque pour que l'utilisation multiple des barrières ne pose aucun problème. □

Exercice 9 (Barrières cumulatives)
On change l'interface de la bibliothèque de l'exercice précédent. Le prototype de la fonction `barrier_wait()` devient

```
int barrier_wait(barrier_t *barrier, int increment);
```

Elle est appelée par un thread qui atteint la barrière. Quand `n_clients` threads ont atteint la barrière, ils sont tous libérés. Le résultat retourné est la somme de tous les `increment` passés par les threads lors de leur appel à `barrier_wait`. □

Exercice 10 (Barrières asymétriques)
On change encore l'interface de la bibliothèque. La fonction `barrier_wait()` est remplacée par trois nouvelles fonctions. L'utilisation des deux fonctions `barrier_init()` et `barrier_destroy()` change légèrement :

```
barrier_t *barrier_init(unsigned n_slaves);
void barrier_destroy(barrier_t *barrier);
int barrier_master(barrier_t *barrier);
int barrier_release(barrier_t *barrier);
int barrier_slave(barrier_t *barrier, int increment);
```

La fonction `barrier_init()` est appelée par le thread maître pour allouer et initialiser une barrière qui sera utilisée avec `n_slaves` threads esclaves.

La fonction `barrier_destroy()` est appelée par le thread maître pour détruire la barrière.

La fonction `barrier_master()` est appelée par le thread maître pour attendre que les `n_slaves` threads esclaves aient atteint la barrière. Quand c'est le cas, `barrier_master()` retourne la somme des incréments donnés par l'ensemble des esclaves.

La fonction `barrier_release()` est appelée par le maître pour libérer l'ensemble des esclaves qui attendent sur la barrière.

La fonction `barrier_slave()` est appelée par un thread esclave qui atteint la barrière. La valeur `increment` est accumulée pour être retournée (`barrier_master()`) au maître. `barrier_slave()` retourne après que le maître a appelé `barrier_release()`. La valeur retournée est la somme des `increment` de l'ensemble des esclaves (i.e. la même valeur que celle retournée au maître par `barrier_master()`).

Question 10.1 Quelle utilité voyez-vous à l'utilisation d'une telle structure de synchronisation ? □

Question 10.2 Donnez une implantation de la bibliothèque `barrier` initiale à l'aide de cette version asymétrique. □

Question 10.3 Donnez une implantation de cette version asymétrique à l'aide de la version initiale de la bibliothèque. □

Question 10.4 Donnez une implantation native de cette version de la bibliothèque. □

Exercice 11 (Verrous récursifs)
On désire implanter une extension des verrous `mutex` de la bibliothèque `pthread` de base. Un de nos `rec_mutex` peut à nouveau être `rec_mutex_lock()` par le thread propriétaire du verrou sans qu'il se bloque. Le propriétaire d'un verrou le relâche par un nombre d'appels à `rec_mutex_unlock()` égal au nombre d'appels à `rec_mutex_lock()` réalisés.

Question 11.1 Définir un type et écrire les fonctions suivantes :

```
typedef struct _rec_mutex_s rec_mutex_t;

struct _rec_mutex_s { ... };

#define REC_MUTEX_INITIALIZER ...
```

```
int rec_mutex_init(rec_mutex_t *rm);
int rec_mutex_destroy(rec_mutex_t *rm);

int rec_mutex_lock(rec_mutex_t *rm);
int rec_mutex_unlock(rec_mutex_t *rm);
```

□

Question 11.2 Définir la fonction :

```
int rec_mutex_trylock(rec_mutex_t *rm);
```

□