TP5 Archi-L2: Threads et Exclusion mutuelle

[Durée: 3 heures]

Omar Rifki - omar.rifki@univ-littoral.fr
modifié du TP de Eric Ramat

27 février 2023

0 Objectif

Ce TP a pour objectif d'étudier la mise en place des techniques d'exclusion mutuelle des processus légers (`thread`) en language C. Nous prendrons example sur le problème du producteur - consommateur vu en cours.

1 Exclusion mutuelle pour 2 threads

Pour rappel, les processus légers (`threads`) sont des processus où les données et les descripteurs de fichiers sont partagés. Leur utilisation est potentiellement problématique sans procédure d'exclusion mutuelle et de synchronisation. Il est donc important de protéger les accès aux ressources partagées à l'aide des sections critiques.

Dans cette première partie, nous allons travailler sur le mécanisme d'exclusion mutuelle de Peterson (slides du cours p.112∼120). Cette version pour deux processus $P0$ et $P1$ est comme suit :

On commence par initialiser le tableau `Veut_entrer` à FAUX et la variable `Tour`.

```
// Initialisation globale
Veut_entrer[0] = FAUX, Veut_entrer[1] = FAUX, Tour = 0
```

Puis pour chaque processus, par exemple $P0$, il y a deux parties :
- le verrouillage de la section critique à l'entrée de celle ci.
- le déverrouillage de la section critique à la sortie.

```
// Entree dans la section critique du processus P0
Veut_entrer[0] = VRAI
Tour = 1
tant que (Veut_entrer[1] est VRAI et Tour = 1) attendre

// Section critique

// Sortie de la section critique de P0
Veut_entrer[0] = FAUX
```

Pour la mise en oeuvre de cet algorithme, nous allons utiliser le code C suivant, où le producteur produit des nombre aléatoires et les place dans un tableau partagé, et le consommateur trie les nombres de ce tableau. Les deux opérations devront être prit en charge séparément par deux processus légers.

```
#define MAX_NUM 30
#define MAX_SIZE 100

int tab[MAX_SIZE];
int size_tab;

void* produce(void* idx) {
    uint* index = (uint*) idx;
    uint count = 0;

    while (count < 5) {
        // debut section critique
        lock(*index);

        tab[size_tab++] = rand() % MAX_NUM;
        printf("produce: %d\n", tab[size_tab - 1]);

        // fin section critique
        unlock(*index);
        count++;
    }
}

void* consume(void* idx) {
    uint* index = (uint*) idx;
    uint count = 0;

    while (count != 5) {
        if (count < size_tab) {
            // debut section critique
            lock(*index);

            // trier et afficher le tableau
            qsort(tab, MAX_SIZE / sizeof(int), sizeof(int), compare);
            printf("consume: sort [%d elements] =>", size_tab);
            for (uint i = 0; i < size_tab; i++)
                printf(" %d", tab[i]);
            printf("\n");
            count = size_tab;

            // fin section critique
            unlock(*index);
        }
    }
}

int compare(const void* a, const void* b) {
    int const* pa = a;
    int const* pb = b;
    return (*pb - *pa);
}
```

La fonction `void* produce()` génère des nombres aléatoires compris entre 0 et `MAX_NUM`, et les place dans le tableau `tab`. L'opération d'insertion dans `tab` est protégée par une section critique.
La fonction `void* consume()` attend que le tableau contienne au moins un élèment nouveau pour appliquer un tri avec la fonction `void qsort()` de `stdlib.h`, qui implémente un algorithme de tri en `O(n log n)`. La phase de tri et d'affichage de `tab` est placée en section critique. Tant que le tableau ne contient pas 5 nombres, on continue l'exécution. la fonction `void qsort()` a besoin d'une fonction en entrée qui compare deux élèments du tableau. `int compare()` donnée dans notre cas nous permet de trier le tableau d'une manière descendante.

Dans le code, nous faisons appel à deux fonctions `lock()` et `unlock()`. La première correspond à l'entrée de la section critique suivant l'algorithme de Peterson, et la deuxième à la sortie de cette section.

Question 1. Dans la fonction `main()` du fichier joint, créer deux `threads`. Nous utiliserons la librairie `pthread` (`POSIX Threads`) pour cela, plus précisement la méthode `pthread_create()`. Ensuite, nous attendrons que les deux processus légers se terminent avec la méthode `pthread_join()`.

Ajouter également les deux fonctions `void lock(uint index)` et `void unlock(uint index)` qui ont comme paramètre le numéro du thread "`index`"". Pour le moment, on les laisse vide. Compiler le code avec la commande suivante :

```
$ gcc -pthread -o prog producer_consumer.c
```

Voici un exemple de sortie en console que l'on peut obtenir :

```
$  ./prog
produce: 13
produce: 16
produce: 27
produce: 25
produce: 23
consume: sort [1 elements] => 13 16 27 25 23
```

Ici, on s'aperçoit que l'opération de tri a eu lieu une fois pour un seul élèment. Le processus `produce` a eu le temps de créer les derniers nombres (16 27 25 23) et les insérer dans le tableau, entre l'exécution de `qsort()` et `count = size_tab` du processus `consume`. Ce qui génère à la fin un résultat faux comme le tableau n'est pas trié.

Question 2. Définir les deux variables `Veut_entrer` et `Tour` en variables globales et ajouter une fonction `init()` qui les initialise au tout début de la fonction `main()`.

Question 3. Définir les fonctions `lock()` et `unlock()` conformément à l'algortihme de Peterson.
Voici un exemple d'exécution à obtenir :

```
$  ./prog
produce: 13
consume: sort [1 elements] => 13
produce: 16
consume: sort [2 elements] => 16 13
produce: 27
consume: sort [3 elements] => 27 16 13
produce: 25
consume: sort [4 elements] => 27 25 16 13
produce: 23
consume: sort [5 elements] => 27 25 23 16 13
```

Remarque. Le le générateur de nombres pseudo-aléatoires `rand()` a besoin d'être initialiser par une valeur imprédictible appelée `seed`, comme le concept d'aléa dans nos machines est simulé par des algorithmes déterministes. Pour cette initialisation, on peut utiliser la fonction `srand()` de `stdlib.h`. Un exemple d'initialisation est :

```
#include <time.h>

srand(time(NULL));
```

Aller plus loin. À la place de l'algorithme de Peterson, vous peut également utiliser le mécanisme d'exclusion mutuelle "mutex" de `pthread`. Pour cela, vous devez définir une variable de type `pthread_mutex_t`, qui sera maniée avec les méthodes :

- `pthread_mutex_init()`
- `pthread_mutex_lock()`
- `pthread_mutex_unlock()`
- `pthread_mutex_destroy()`

2 Exclusion mutuelle pour N threads

À présent, nous allons définir 5 threads qui produisent des nombres aléatoires. Les fonctions `lock()` et `unlock()` doivent être adaptées suivant l'algotithme de Peterson généralisé. Pour rappel :

```
// Initialisation globale
in_stage : tableau de N entiers initialises a -1
last_process : tableau de N entiers initialises a 0

// Entree dans la section critique du processus Pi
Pour j de 0 a N
  in_stage[i] = j
  last_process[j] = i
  Pour k de 0 a N
    Si k != i
      Tant que in_stage[k] >= in_stage[i] ET last_process[j] = i
        attendre

// Section critique

// Sortie de la section critique
in_stage[i] = -1
```

Question 4. Copier le fichier `producer_consumer.c` précédent. Dans le nouveau fichier, modifier la fonction `main()` afin de lancer 5 `threads` de type `produce`. Attention aux numéros des threads.

Remplacer le 5 par 25 dans `while (count != 5)`, et le 30 par 50 pour `MAX_NUM`.

Question 5. Modifier les fonctions `init()`, `lock()` et `unlock()`. Les variables `Veut_entrer` et `Tour` sont remplacées par `in_stage` et `last_process`.

Voici un exemple d'exécution à obtenir :

```
$  ./prog
produce: 27
produce: 41
produce: 48
produce: 46
produce: 46
produce: 26
produce: 45
produce: 24
produce: 4
consume: sort [9 elements] => 48 46 46 45 41 27 26 24 4
produce: 25
produce: 42
produce: 16
produce: 38
produce: 5
consume: sort [14 elements] => 48 46 46 45 42 41 38 27 26 25 24 16 5 4
produce: 31
produce: 26
produce: 33
produce: 19
produce: 11
consume: sort [19 elements] => 48 46 46 45 42 41 38 33 31 27 26 26 25 24
 19 16 11 5 4
produce: 47
produce: 44
produce: 8
produce: 41
consume: sort [23 elements] => 48 47 46 46 45 44 42 41 41 38 33 31 27 26
 26 25 24 19 16 11 8 5 4
produce: 8
produce: 18
consume: sort [25 elements] => 48 47 46 46 45 44 42 41 41 38 33 31 27 26
 26 25 24 19 18 16 11 8 8 5 4
```