Systèmes d'exploitation

TP 5 - Threads et verrous

2023-05-11

> **API des threads**
>
> Avant toute chose, lisez le chapitre Thread API de l'ouvrage de référence *Operating Systems: Three Easy Pieces*. Celui-ci vous fournira une bonne référence de base pour utiliser les threads, verrous et condition variables.

1 Exercice : prise en main de l'API thread d'Unix

Le but de cet exercice est d'écrire un programme multi-threads qui calcule la somme S suivante en utilisant un tableau contenant les valeurs de i et des threads.

$$S = \sum_{i=1}^{N} i^2$$

1. Commençons par créer un tableau d'entiers avec `N` éléments. On considère que `N = 1000`.

```c
#include <stdlib.h>

#define N 1000

int array[N];

int main() {
    for (int i = 0; i < N; i++) {
        array[i] = i + 1;
    }
    // ...
    return EXIT_SUCCESS;
}
```

2. À vous de jouer à présent :

   - Divisez le tableau en K parties égales (K < 10).
   - Créez K threads qui calculent la somme du carré des éléments de chaque partie. Pour cela, utilisez la fonction `pthread_create()`.
   - Utilisez le thread principal pour attendre que tous les threads terminent leur exécution et rassembler les résultats. La fonction `pthread_join()` vous permettra attendre la fin de chaque thread et de récupérer son résultat. Vous n'avez pas besoin d'utiliser de verrous dans cet exercice.
   - Affichez la somme totale et vérifiez que $\sum_{i=1}^{1000} i^2 = 333833500$.

> **i** ***I/O bound* et *CPU bound***
>
> Schématiquement, un programme informatique (ou une partie d'un programme) est :
>
> - soit ***I/O bound*** : le temps nécessaire pour l'exécuter est déterminé principalement par la vitesse des opérations d'entrées/sorties (*I/O*) ;
> - soit ***CPU bound*** : le temps d'exécution est déterminé principalement par la vitesse du processeur.
>
> Dans le cas d'un programme *CPU bound*, il peut être intéressant d'augmenter la puissance de calcul CPU à disposition du programme en parallélisant l'exécution sur plusieurs processeurs.

3. À votre avis, votre programme est-il *I/O bound* ou *CPU bound* ? En sachant cela, quel serait intuitivement le nombre idéal de threads à utiliser pour faire en sorte que notre programme s'exécute le plus rapidement possible ?
4. À présent fixez la valeur de N à `10000`. Quel résultat obtenez-vous ? Que se passe-t-il ?

2 Exercice : verrous

Modifiez le programme de l'exercice précédent, faites en sorte que les fonctions des threads ne retournent pas de valeur. À la place, les threads accèdent à une même variable somme partagée qu'ils modifient à chaque fois qu'ils calculent le carré d'un élément de la liste. Il en résultent une **section critique** qui doit être protégée par un pthread_mutex.

Vérifiez que vous obtenez toujours le même résultat que précédemment pour `N = 1000`.

3 Exercice : structures de données synchronisées

L'ajout de verrous à une structure de données pour la rendre utilisable de façon concurrente par plusieurs threads rend la structure ***"thread safe"***. Le but du présent exercice est de construire quelques structures de données synchronisées, c'est-à-dire *thread safe*.

1. On considère le code d'un compteur non-synchronisé ci-dessous. Vous devez utiliser un verrou aux endroits opportuns pour permettre à plusieurs threads d'utiliser de façon concurrente ce compteur tout en évitant les problèmes de *race conditions*. Pour cela, vous aurez besoin de `pthread_mutex`.

```c
typedef struct __counter_t {
    int value;
    // pthread_mutex_t lock;
} counter_t;

void init(counter_t *c) {
    c->value = 0;
}

void increment(counter_t *c) {
    c->value++;
}

int get(counter_t *c) {
    return c->value;
}
```

2. Une fois que vous avez rendu le compteur synchronisé, écrivez un programme qui comprend plusieurs threads qui incrémentent le compteur simultanément, en vous inspirant du programme de l'exercice 1. Vérifiez que le compteur prend bien la valeur attendue.

3. Intéressons-nous maintenant à implémentation d'une liste chaînée synchronisée à partir du code ci-dessous. Cette liste ne stocke que des nombres entiers et il est uniquement possible d'ajouter des éléments à la liste ou de vérifier qu'un élément est présent. Comme pour le compteur, vous devez utiliser un verrou à différents endroits pour rendre cette liste *thread safe*.

```c
#include <stdio.h>

typedef struct __node_t {
    int key;
    struct __node_t *next;
} node_t;

typedef struct __list_t {
    node_t *head;
    // pthread_mutex_t lock;
} list_t;

void List_Init(list_t *L) {
    L->head = NULL;
```

```
}

int List_Insert(list_t *L, int key) {
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        return -1; // fail
    }

    new->key = key;
    new->next = L->head;
    L->head = new;
    return 0; // success
}

int List_Lookup(list_t *L, int key) {
    node_t *curr = L->head;
    while (curr) {
        if (curr->key == key) {
            return 0; // success
        }
        curr = curr->next;
    }
    return -1; // failure
}
```

4. Écrivez maintenant un programme multi-threads qui utilise votre liste synchronisée.

À retenir

- La fonction **pthread_create()** démarre un nouveau thread dans le processus appelant.
- La fonction **pthread_join()** permet d'attendre qu'un thread donné se termine.
- Si un programme est ***I/O bound*** le temps nécessaire pour l'exécuter est déterminé principalement par la vitesse des opérations d'entrées/sorties (I/O).
- Si un programme est ***CPU bound*** : le temps d'exécution est déterminé principalement par la vitesse du processeur.
- Les **pthread_mutex** permettent de synchroniser l'accès entre des variables partagées par des threads.
- L'ajout de verrous à une structure de données pour la rendre utilisable de façon concurrente par plusieurs threads rend la structure ***"thread safe"***