Systèmes d'exploitation

TP 5 - Threads et verrous - Correction

2023-05-11

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 $\sum_{i=1}^{1000} i^2 = 333833500$.

```c
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>

#define N 1000
#define K 5

typedef struct { int id; } myarg_t;
typedef struct { int sum; } myret_t;

int array[N];

void *mythread(void *arg) {
    int id = ((myarg_t *)arg)->id;
    int start = id * (N / K);
    int end = (id + 1) * (N / K);

    myret_t *ret = malloc(sizeof(myret_t));
    ret->sum = 0;

    for (int i = start; i < end; i++) {
        ret->sum += array[i] * array[i];
    }

    printf("id = %d, start = %d, end = %d, sum = %d\n", id, start, end,
 ↪  ret->sum);
    return (void *) ret;
}


int main() {
    for (int i = 0; i < N; i++) {
        array[i] = i + 1;
    }

    pthread_t threads[K];
    myarg_t args[K];
    for (int i = 0; i < K; i++) {
        args[i] = (myarg_t ){i};
        pthread_create(&threads[i], NULL, mythread, &args[i]);
    }

    int sum = 0;
```

```
    for (int i = 0; i < K; i++) {
        myret_t *ret;
        pthread_join(threads[i], (void **)&ret);
        sum += ret->sum;

        free(ret);
    }

    printf("S = %d\n", sum);
    return EXIT_SUCCESS;
}
```

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 ?

Notre programme est *CPU bound* : il ne réalise pas d'entrées/sorties et se contente de calculer des carrés de nombres. Intuitivement, on voudrait que notre programme utilise autant de threads qu'il y a de processeurs sur la machine pour paralléliser au maximum les calculs à effectuer.

4. À présent, fixez la valeur de N à `10000`. Quel résultat obtenez-vous ? Que se passe-t-il ?

On obtient un résultat négatif. C'est le signe d'un dépassement d'entier : une opération mathématique a produit une valeur numérique supérieure à celle représentable dans l'espace de stockage disponible.

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`.

```
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>

#define N 1000
#define K 5

typedef struct { int id; } myarg_t;

int array[N];
```

```c
int sum = 0;

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *mythread(void *arg) {
    int id = ((myarg_t *)arg)->id;
    int start = id * (N / K);
    int end = (id + 1) * (N / K);

    for (int i = start; i < end; i++) {
        pthread_mutex_lock(&lock);
        sum += array[i] * array[i];
        pthread_mutex_unlock(&lock);
    }

    return NULL;
}


int main() {
    for (int i = 0; i < N; i++) {
        array[i] = i + 1;
    }

    pthread_t threads[K];
    myarg_t args[K];
    for (int i = 0; i < K; i++) {
        args[i] = (myarg_t ){i};
        pthread_create(&threads[i], NULL, mythread, &args[i]);
    }

    for (int i = 0; i < K; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("S = %d\n", sum);
    return EXIT_SUCCESS;
}
```

Notez que plutôt qu'ajouter le carré à la variable globale `sum`, on pourrait calculer une somme partielle des carrés par thread et ajouter cette valeur à la somme globale à la fin de la function `mythread`. Cela diminuerait le nombre d'opérations d'acquisition/libération du verrou et pourrait améliorer les performances.

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
#include <pthread.h>

typedef struct __counter_t {
    int value;
    pthread_mutex_t lock;
} counter_t;

void init(counter_t *c) {
    c->value = 0;
    pthread_mutex_init(&c->lock, NULL);
}

void increment(counter_t *c) {
    pthread_mutex_lock(&lock);
    c->value++;
    pthread_mutex_unlock(&lock);
}

int get(counter_t *c) {
    pthread_mutex_lock(&lock);
    int rc = c->value;
    pthread_mutex_unlock(&lock);
    return rc;
}
```

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.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define K 5
```

```c
// ...

counter_t counter;

void *mythread(void *args) {
    for (int i = 0; i < 100; i++) {
        increment(&counter);
    }

    return NULL;
}


int main() {
    init(&counter);

    pthread_t threads[K];
    for (int i = 0; i < K; i++) {
        pthread_create(&threads[i], NULL, mythread, NULL);
    }

    for (int i = 0; i < K; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("counter = %d\n", get(&counter));
    return EXIT_SUCCESS;
}
```

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 <pthread.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;
    pthread_mutex_init(&L->lock, NULL);
}

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

    new->key = key;
    new->next = L->head;
    L->head = new;
    pthread_mutex_unlock(&L->lock);
    return 0; // success
}

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

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

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define N 1000
#define K 5
```

```c
// ...

list_t list;

void *mythread(void *arg) {
    int id = *(int *) arg;
    int start = id * (N / K);
    int end = (id + 1) * (N / K);

    for (int i = start; i < end; i++) {
        List_Insert(&list, i);
    }

    return NULL;
}


int main() {
    List_Init(&list);

    pthread_t threads[K];
    int args[K];
    for (int i = 0; i < K; i++) {
        args[i] = i;
        pthread_create(&threads[i], NULL, mythread, &args[i]);
    }

    for (int i = 0; i < K; i++) {
        pthread_join(threads[i], NULL);
    }

    char *isPresent = List_Lookup(&list, N - 1) == 0 ? "true" : "false";
    printf("%d is present in list: %s\n", N - 1, isPresent);
    return EXIT_SUCCESS;
}
```