TP3 - Thread & Synchronisation

Système d'exploitation - Sergent Pierre-Louis LSI-1

Sommaire

Introduction

```
.
+-- compiled
|      tp3
+-- headers
|      threads.h
+-- src
|      threads.c
+-- main.c
+-- Makefile
+-- README.md
+-- TP3_thread_synchro_L3_ASYRIA_2017_18.pdf
```

L'organisation du TP3 se fait comme telle, avec trois dossiers:

- compiled : fichiers exécutables
- hearders : regroupe tous les headers contenant les définitions de fonctions et de structures
- src : regroupe l'implémentation des fonctions

A la racine on retrouve le `main.c` qui est le point d'entré du projet.

La compilation et l'exécution se fait à travers une `Makefile`.

```
tp3: main.c ./headers/threads.h
    gcc -g main.c ./src/threads.c -o ./compiled/tp3 -lpthread

run: main.c
    ./compiled/tp3
```

Permettant la compilation grâce à la commande `make` et l'exécution avec la commande `make run`.

Tout au long du TP, les threads sont lancés depuis la fonction `main`, les fonctions utilisées pour ces derniers sont définies dans le header : `threads.h` et implémentés dans `threads.c` se trouvant dans le dossier `./src/`.

Question 1

Header

```
void *fct_thread(void *args);

typedef struct fct_threads_args {
    int *global;
    int local;
} fct_threads_args_t;
```

Nous avons donc une fonction qui sera utilisé par notre thread pour modifier la variable globale.

En dessous se trouve la `struct fct_threads_args` qui sert à définir les paramètres à passer dans la fonction `fct_threads_args`. Si on a plus d'un argument à passer alors la mise en place d'une telle structure est nécessaire et nous en retrouvons donc tout au long du TP.

Ici simplement en paramètre nous avons: - `global` : un pointeur vers la variable globale qui sera modifiée par les trois threads - `local` : variable locale qui sera modifiée dans la fonction du thread

Implémentation

```
void *fct_thread(void *args) {
    fct_threads_args_t * arguments = args;
    int *global = arguments->global;
    int local = arguments->local;

    *global += 11;
    local += 1;
    printf("Local variable thread %d : %d\n", local/10, local);
    pthread_exit(NULL);
}
```

Les trois première ligne de la fonction servent donc à récupérer les paramètres qui ont été passés grâce à la structure `fct_threads_args_t`.

Appel du **main**

```
// Question 1
printf("\nQuestion 1:\n");

pthread_t thread1, thread2, thread3;
fct_threads_args_t func_args1, func_args2, func_args3;

int global_var = 100;
int iret1, iret2, iret3;

func_args1.global = &global_var;
func_args1.local = 10;

func_args2.global = &global_var;
func_args2.local = 20;

func_args3.global = &global_var;
func_args3.local = 30;

iret1 = pthread_create(&thread1, NULL, fct_thread, (void*) &func_args1);
iret2 = pthread_create(&thread2, NULL, fct_thread, (void*) &func_args2);
iret3 = pthread_create(&thread3, NULL, fct_thread, (void*) &func_args3);

pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
pthread_join(thread3, NULL);
printf("Global variable : %d\n", global_var);
```

Output

```
Question 1:
Local variable thread 1 : 11
Local variable thread 2 : 21
Local variable thread 3 : 31
Global variable : 133
```

Les outputs sont conformes aux résultats attendus.

Question 2

Header

```
void *synchro_thread(void *args);

typedef struct synchro_thread_args {
    char **sentence;
    pthread_mutex_t *lock1;
    pthread_mutex_t *lock2;
```

```
    int i;
} synchro_thread_args_t;
```

Pour cette question j'ai décidé d'utiliser uniquement une fonction pour les deux threads, un paramètre nous permettant de différencier les deux threads.

Pour la structure on retrouve :

- `sentence` : tableau de string contenant les mots à afficher
- `lock1` & `lock2`: les deux locks des mutex nous permettant de bloquer un thread lors d'une opération puis de débloquer l'autre thread afin qu'il réalise son action à son tour. Cela nous permet de synchroniser les threads
- `i` : entier permettant de savoir quel thread estr exécute la fonction `synchro_thread` et ainsi adapter son comportement.

Implémentation

```
void *synchro_thread(void *args) {
    synchro_thread_args_t *arguments = args;
    char **sentence = arguments->sentence;
    pthread_mutex_t *lock1 = arguments->lock1;
    pthread_mutex_t *lock2 = arguments->lock2;
    int i = arguments->i;

    if (i == 0) {
        for (int j=0; j < 4; j++) {
            if (j % 2 == 0) {
                pthread_mutex_lock(lock1);
                printf("%s ", sentence[j]);
                pthread_mutex_unlock(lock2);
            }
        }
        pthread_exit(NULL);
    } else {
        for (int j=0; j < 4; j++) {
            if (j % 2 == 1 && j != 3) {
                pthread_mutex_lock(lock2);
                printf("%s ", sentence[j]);
                pthread_mutex_unlock(lock1);
            } else if (j == 3) {
                pthread_mutex_lock(lock2);
                printf("%s ", sentence[j]);
            }
        }
        printf("\n");
        pthread_exit(NULL);
    }
}
```

Appel du **main**

```
// Question 2
printf("\nQuestion 2:\n");

pthread_t thread1, thread2;
pthread_mutex_t lock1, lock2;
synchro_thread_args_t func_args1, func_args2;

char *sentence[100] = {"je", "synchronise", "mes", "threads"};

func_args1.sentence = sentence;
func_args1.lock1 = &lock1;
func_args1.lock2 = &lock2;
func_args1.i = 0; // even

func_args2.sentence = sentence;
func_args2.lock2 = &lock2;
func_args2.lock1 = &lock1;
func_args2.i = 1; // odd

pthread_mutex_init(&lock1, NULL);
pthread_mutex_init(&lock2, NULL);
pthread_mutex_lock(&lock2);

pthread_create(&thread2, NULL, synchro_thread, (void*) &func_args2);
pthread_create(&thread1, NULL, synchro_thread, (void*) &func_args1);

pthread_join(thread1, NULL);
pthread_join(thread2, NULL);

pthread_mutex_destroy(&lock1);
pthread_mutex_destroy(&lock2);
```

Output

```
Question 2:
je synchronise mes threads
```

L'output est conforme au résultat attendu.

Question 3

Header

```
void *spleeping_thread(void *args);
void *barrier_thread(void *args);

typedef struct spleeping_thread_args {
    pthread_barrier_t *barrier;
```

```
    int i;
} spleeping_thread_args_t;

typedef struct barrier_thread_args {
    pthread_barrier_t *barrier;
    int n;
} barrier_thread_args_t;
```

On retrouve ici les deux fonctions `sleeping_thread` et `barrier_thread`. La première va être utilisé pour la création des `n` thread et la deuxième servira de thread barrière. Le but étant de coordonner tous les threads en attendant que tous les `sleeping_thread` aient fini de s'exécuter.

Implémentation

```
void *spleeping_thread(void *args) {
    spleeping_thread_args_t *arguments = args;
    pthread_barrier_t *barrier = arguments->barrier;
    int i = arguments->i;

    srand(time(NULL));
    int random = rand() % 3;
    sleep(random);

    printf("[ thread %d ] Point atteint\n", i);
    pthread_barrier_wait(barrier);
    pthread_exit(NULL);
    return 0;
}

void *barrier_thread(void *args) {
    barrier_thread_args_t *arguments = args;
    pthread_barrier_t *barrier = arguments->barrier;
    int num_threads = arguments->n;

    pthread_barrier_init(barrier, NULL, num_threads);

    srand(time(NULL));
    int random = rand() % 3;
    sleep(random);

    printf("[ b_thread ] Point atteint, en attente...\n");

    pthread_barrier_wait(barrier);
    pthread_barrier_destroy(barrier);
    printf("[ b_thread ] Je peux continuer !\n");
    pthread_exit(NULL);
    return 0;
}
```

Une fois que les `sleeping_thread` ont atteint un certain point ils attendent que tous les autres threads se terminent et ce grâce à la ligne :

```
pthread_barrier_wait(barrier);
```

Le `barrier_thread` est initialisé de la sorte :

```
pthread_barrier_init(barrier, NULL, num_threads);
```

puis il attend lui même que les `sleeping_thread` aient fini leurs tâches. A la fin desquelles on peut détruire la barrière et faire continuer le `barrier_thread`.

```
pthread_barrier_wait(barrier);
pthread_barrier_destroy(barrier);
printf("[ b_thread ] Je peux continuer !\n");
```

Appel du **`main`**

```
// Question 3
printf("\nQuestion 3:\n");
int n = 4;

pthread_t threads[n], b_thread;
pthread_barrier_t barrier;
barrier_thread_args_t barrier_args;
spleeping_thread_args_t thread_args[n];

barrier_args.barrier = &barrier;
barrier_args.n = n;

pthread_create(&b_thread, NULL, barrier_thread, (void*) &barrier_args);

for (int i=1; i < n; i++) {
    thread_args[i].barrier = &barrier;
    thread_args[i].i = i;
    pthread_create(&threads[i], NULL, spleeping_thread, (void*) &thread_args[i]);
}

for (int j=1; j < n; j++) {
    pthread_join(threads[j], NULL);
}
pthread_join(b_thread, NULL);
```

Output

```
Question 3:
[ b_thread ] Point atteint, en attente...
[ thread 3 ] Point atteint
[ thread 2 ] Point atteint
[ thread 1 ] Point atteint
[ b_thread ] Je peux continuer !
```

L'output est conforme au résultat attendu.

Question 4

Header

```
void *reader_thread(void *args);
void *writer_thread(void *args);

typedef struct reader_thread_args {
    pthread_mutex_t *lock_r;
    pthread_mutex_t *lock_w;
    int **buffer;
    int index;
} reader_thread_args_t;

typedef struct writer_thread_args {
    pthread_mutex_t *lock_r;
    pthread_mutex_t *lock_w;
    int **buffer;
    int index;
    int data;
} writer_thread_args_t;
```

`reader_thread_args` :

- `lock_r` : lock du lecteur
- `lock_w` : lock du modificateur
- `buffer` : tampon qui est un tableau d'entier
- `index` : position dans le tableau de la valeur à lire

`writer_thread_args` :

- `lock_r` : lock du lecteur
- `lock_w` : lock du modificateur
- `buffer` : tampon qui est un tableau d'entier
- `index` : position dans le tableau de la valeur à modifier
- `data` : nouvelle valeur pour l'index indiqué

Implémentation

```
void read_buffer(int index, int **buffer) {
    printf("buffer[%d] = %d\n", index, (*buffer)[index]);
}

void write_buffer(int index, int value, int **buffer) {
    (*buffer)[index] = value;
    printf("modified value at index %d : buffer[%d] = %d\n", index, index, (*buffer)[index]);
}

void *reader_thread(void *args) {
    reader_thread_args_t *arguments = args;
    int **buffer = arguments->buffer;
    pthread_mutex_t *lock_r = arguments->lock_r;
```

```
    pthread_mutex_t *lock_w = arguments->lock_w;
    int i = arguments->index;

    printf("Reading...\n");

    pthread_mutex_lock(lock_r);
    read_buffer(i, buffer);
    pthread_mutex_unlock(lock_w);

    // See result after write
    pthread_mutex_lock(lock_r);
    read_buffer(i, buffer);
    pthread_mutex_unlock(lock_w);
    pthread_exit(NULL);
}

void *writer_thread(void *args) {
    writer_thread_args_t *arguments = args;
    int **buffer = arguments->buffer;
    pthread_mutex_t *lock_r = arguments->lock_r;
    pthread_mutex_t *lock_w = arguments->lock_w;
    int i = arguments->index;
    int value = arguments->data;

    printf("Writing...\n");

    pthread_mutex_lock(lock_w);
    write_buffer(i, value, buffer);
    pthread_mutex_unlock(lock_r);
    pthread_exit(NULL);
}
```

De manière similaire à la question 2, nous utilisons les locks pour nous assurer que l'on lit d'abord la valeur, puis on l'a modifie, avant de la lire à nouveau pour vérifier si les changements ont été effectué.

Appel du **main**

```
// Question 4
printf("\nQuestion 4:\n");

pthread_t t_reader, t_writer;
pthread_mutex_t lock_r, lock_w;
reader_thread_args_t reader_args;
writer_thread_args_t writer_args;

int *buffer = (int*)malloc(10*sizeof(int));
for (int i=0; i < 10; i++) {
    buffer[i] = i;
}

reader_args.lock_r = &lock_r;
```

```
reader_args.lock_w = &lock_w;
reader_args.buffer = &buffer;
reader_args.index = 4;

writer_args.lock_r = &lock_r;
writer_args.lock_w = &lock_w;
writer_args.buffer = &buffer;
writer_args.index = 4;
writer_args.data = 100;

pthread_mutex_init(&lock_r, NULL);
pthread_mutex_init(&lock_w, NULL);
pthread_mutex_lock(&lock_w);

pthread_create(&t_reader, NULL, reader_thread, (void*) &reader_args);
pthread_create(&t_writer, NULL, writer_thread, (void*) &writer_args);

pthread_join(t_reader, NULL);
pthread_join(t_writer, NULL);

pthread_mutex_destroy(&lock_r);
pthread_mutex_destroy(&lock_w);
```

Le `buffer` est donc un tableau d'entier qui vont de 1 à 9. Nous allons donc lire et modifier la valeur d'indice 4.

Output

```
Question 4:
Reading...
buffer[4] = 4
Writing...
modified value at index 4 : buffer[4] = 100
buffer[4] = 100
```

L'output est conforme au résultat attendu.