Threads, Mutex et Programmation Concurrente en C

1. Introduction à la Programmation Concurrente

La **programmation concurrente** permet à un programme d'effectuer plusieurs tâches simultanément, contrairement à la programmation séquentielle où les opérations se font une par une.

Avantages :

- Amélioration des performances
- Meilleure utilisation des ressources
- Réactivité accrue des applications

Trois méthodes d'implémentation :

- Les processus
- Les threads (notre focus)
- Le multiplexage

2. Qu'est-ce qu'un Thread ?

Un **thread** (fil d'exécution) est une suite logique d'instructions à l'intérieur d'un processus, gérée automatiquement par le noyau du système d'exploitation.

Caractéristiques des threads :

Contexte propre :

- Identifiant unique
- Pile d'exécution (stack)
- Pointeur d'instruction
- Registre de processeur

Ressources partagées :

- Même espace d'adressage virtuel
- Même code
- Même heap
- Mêmes bibliothèques partagées
- Mêmes descripteurs de fichiers

Avantages des threads vs processus :

- Création plus rapide
- Commutation plus efficace
- Communication plus facile (mémoire partagée)
- Pas de hiérarchie père-fils stricte

3. Utilisation des Threads POSIX

Compilation

bash

```
gcc -pthread main.c
```

Créer un thread

c

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
```

Paramètres :

- `thread` : pointeur vers l'identifiant du thread
- `attr` : attributs du thread (généralement `NULL`)
- `start_routine` : fonction à exécuter
- `arg` : argument à passer à la fonction

Récupérer un thread

c

```
int pthread_join(pthread_t thread, void **retval);
```

Détacher un thread

c

```
int pthread_detach(pthread_t thread);
```

Exemple pratique

c

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

void *thread_routine(void *data) {
    pthread_t tid = pthread_self();
    printf("Thread [%ld]: Message du thread\n", tid);
    return NULL;
}

int main(void) {
    pthread_t tid1, tid2;

    // Création des threads
    pthread_create(&tid1, NULL, thread_routine, NULL);
    pthread_create(&tid2, NULL, thread_routine, NULL);

    // Attendre la fin des threads
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    return 0;
}
```

4. Problèmes de Synchronisation

Le Data Race

Quand plusieurs threads accèdent simultanément à la même donnée, cela peut créer des **erreurs de synchronisation**.

Exemple problématique :

c

```c
// Variable partagée
unsigned int count = 0;

// Dans chaque thread
count++; // Opération non-atomique !
```

Problème : L'opération `count++` n'est pas atomique :

1. Lecture de la valeur
2. Incrémentation
3. Sauvegarde

Si deux threads font cela simultanément, des valeurs peuvent être perdues.

5. Les Mutex (Mutual Exclusion)

Un **mutex** est un verrou qui permet de réguler l'accès aux ressources partagées.

Déclaration et initialisation

c

```c
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
```

Verrouillage et déverrouillage

c

```c
pthread_mutex_lock(&mutex);    // Verrouiller
// Section critique
pthread_mutex_unlock(&mutex);  // Déverrouiller
```

Destruction

c

```c
pthread_mutex_destroy(&mutex);
```

Exemple avec mutex

c

```c
typedef struct {
    pthread_mutex_t count_mutex;
    unsigned int count;
} t_counter;

void *thread_routine(void *data) {
    t_counter *counter = (t_counter *)data;

    for (int i = 0; i < 10000; i++) {
        pthread_mutex_lock(&counter->count_mutex);
        counter->count++;
        pthread_mutex_unlock(&counter->count_mutex);
    }
    return NULL;
}
```

6. Attention aux Deadlocks

Un **deadlock** (interblocage) survient quand des threads s'attendent mutuellement.

Exemple de deadlock :

c

```c
// Thread 1
pthread_mutex_lock(&mutex_A);
pthread_mutex_lock(&mutex_B); // Attend mutex_B

// Thread 2
pthread_mutex_lock(&mutex_B);
pthread_mutex_lock(&mutex_A); // Attend mutex_A
```

Solutions pour éviter les deadlocks :

1. **Ordre strict** : toujours acquérir les mutex dans le même ordre
2. **Timeout** : utiliser des timeouts sur les verrous
3. **Détection** : implémenter un système de détection
4. **Évitement** : analyser l'état avant d'allouer

7. Bonnes Pratiques

Test et débogage

bash

```bash
# Compilation avec détection de data race
gcc -fsanitize=thread -g programme.c

# Utilisation de Valgrind
valgrind --tool=helgrind ./programme
valgrind --tool=drd ./programme
```

Conseils :

- Tester plusieurs fois de suite
- Minimiser les sections critiques
- Éviter les mutex imbriqués
- Libérer les ressources correctement
- Utiliser les outils de détection d'erreurs

8. Résumé

Les threads permettent :

- L'exécution parallèle de tâches
- Une meilleure utilisation des ressources
- Une communication efficace via la mémoire partagée

Les mutex protègent :

- Les accès concurrents aux données
- L'intégrité des opérations critiques
- Contre les data races

Points d'attention :

- Toujours initialiser et détruire les mutex
- Éviter les deadlocks par un design soigneux
- Tester rigoureusement les programmes concurrents
- Utiliser les outils de détection d'erreurs

La programmation concurrente est puissante mais délicate. Une compréhension solide des threads et mutex est essentielle pour développer des applications robustes et performantes.