Programmation avancée

TD12 : Concurrence et Parallélisme

Jacques-Henri Jourdan Armaël Guéneau Arnaud Golfouse

Pensez bien à vérifier que votre code compile. Si vous n'avez pas terminé, remplacez les parties manquantes par `todo!()`.

Pensez également à tester votre implémentation avec `cargo test`.

Note Téléchargez le code de démarrage de ce TP à l'adresse suivante : `https://jhjourdan.gitlabpages.inria.fr/prog3-l3-ensps/tds/12-concurrence/td12.zip`

1 Une 'base de données' concurrente

Pour expliquer les difficultés de la concurrence nous allons implémenter une 'base de données' en utilisant une `HashMap` et des `RwLock`. Le type `RwLock` généralise les mutex en permettant à plusieurs threads de lire la valeur *ou* à un thread d'écrire une valeur. Il suffit de traiter cela comme un mutex avec une fonction `read` et `write` qui déclenchent le bon type de verrouillage.

Remarques:

Pour ce TP vous pouvez utiliser la fonction `unwrap` pour déplier le `LockResult` renvoyé par les fonctions `write` et `read`. Ce résultat est utilisé pour gérer le 'lock poisoning', un détail de l'implémentation de la librairie standard de Rust qui n'est pas le sujet d'aujourd'hui.

—

Pour aider à la détection de bugs de concurrence, on fournit une implémentation alternative de `RwLock`, importable avec `crate::testing_rwlock::RwLock`.

Nous utiliserons le type `Database` ci-dessous. On utilise deux niveaux de locks, un autour du `HashMap` pour contrôler les insertions et délétions, et un autour de chaque valeur pour contrôler les lectures et écritures. Cela veut dire que plusieurs threads doivent être capables de lire et écrire simultanément, mais qu'un seul thread à la fois peut ajouter ou effacer des valeurs.

```rust
struct Database {
    inner: RwLock<HashMap<String, RwLock<u64>>>,
}

impl Database {
    pub fn new() -> Self { todo!() }

    pub fn get(&self, key: &String) -> Option<u64> { todo!() }

    pub fn set(&self, key: &String, value: u64) -> bool { todo!() }

    pub fn insert(&self, k: String, v: u64) -> Option<u64> { todo!() }

    pub fn delete(&self, key: &String) -> Option<u64> { todo!() }
}
```

Exercice 1. *Implémenter `get`, qui doit renvoyer la valeur associée à la clé si elle est présente et `None` sinon. Plusieurs appels à `get` doivent être capable de s'exécuter en parallèle.*

Exercice 2. *Implémenter `set`, qui doit écrire la valeur associé à la clé si elle est présente. Cette fonction renvoie `true` si l'écriture était possible et `false` sinon. Plusieurs appels à `set` avec des clés différentes doivent être capable de s'exécuter en parallèle.*

Exercice 3. *Implémenter `insert`, qui doit ajouter la valeur associée à la clé. Si la clé n'est pas encore présente, la fonction renvoie `None`, sinon la fonction renvoie l'ancienne valeur associée à la clé.*

Exercice 4. *Implémenter `delete`, qui doit enlever la clé de la table de hachage. Cette fonction renvoie `Some(v)` si la clé était associée à une valeur `v`, et `None` sinon.*

Exercice 5. *Nous allons maintenant ajouter une opération un peu spéciale: `fn swap(&self, k1: &String, k2: &String)` qui échange les valeurs de deux clés. Si une des valeurs n'existe pas dans la base de données, `swap` devra échouer. Il faut que plusieurs `swap` puissent être effectués en parallèle.*

Exercice 6. *Vérifier que votre code réussisse le test `deadlock`. Le corriger sinon. Quelle est la source du problème ?*

(Remarque: lancez le test avec la commande `cargo test -- --ignored deadlock`)

2 Fork-Join Parallelism

Le code source fourni avec le sujet inclut une implémentation récursive de `quicksort`. Dans cet exercise nous allons la rendre parallèle.

La fonction `std::thread::spawn` de la librairie standard de Rust nous permet de créer des nouveaux threads et d'y lancer des calculs facilement.

Exercice 7. *Implémentez la fonction* `quicksort_par1`.

On utilisera la fonction `spawn` *pour paralléliser les appels récursifs. Si les données d'entrées sont moins d'une longueur critique (*`MIN_LENGTH = 64`*), ne plus faire d'appels parallèles.*

En Rust, nous travaillons souvent avec des *slices* (`&[T]` / `&mut[T]`) plutôt que des vecteurs. Une slice emprunte les permissions d'un vecteur ou un tableau et nous permet de lire ou écrire des valeurs mais pas désallouer ou agrandir la collection sous-jacente.

Exercice 8. *On aimerait changer notre fonction de tri pour prendre une slice* `&mut [T]` *en entrée, mais si on change le type on constate une erreur du borrow checker, que veut-elle dire? Ou se trouve le problème et que faudrait-il changer?*

Pour rectifier ce problème nous allons utiliser une API plus avancée de la librairie standard: `std::thread::scope`, qui a le type suivant:

```
pub fn scope<'env, F, T>(f: F) -> T
where
    F: for<'scope> FnOnce(&'scope Scope<'scope, 'env>) -> T;

impl<'scope, 'env> Scope<'scope, 'env> {
    pub fn spawn<F, T>(&'scope self, f: F) -> ScopedJoinHandle<'scope, T>
    where
        F: FnOnce() -> T + Send + 'scope,
        T: Send + 'scope;
}
```

Note: le `for` quantifie universellement sur les lifetimes, ici demandant que `F` soit `FnOnce` pour n'importe quelle lifetime.

Exercice 9. *Expliquer pourquoi cet API nous permettrait de partager des emprunts, et sous quelles circonstances.*

Exercice 10. *Implémentez la fonction* `quicksort_par2`.

On adaptera la solution utilisée dans `quicksort_par1` *pour utiliser* `std::thread::scope`. *Au passage, convertir votre solution pour utiliser des* `&mut [T]`. *Vous aurez besoin de la fonction* `split_at_mut`.