Programmation avancée

TD 11 : Concurrence et Parallélisme

Jacques-Henri Jourdan Armaël Guéneau Arnaud Golfouse Paul Robert

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/11-concurrence/td11.zip`

1 Fork-Join Parallelism

Fichier `src/quicksort.rs`

Le code source fourni avec le sujet inclut une implémentation récursive de `quicksort`. Dans cet partie 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 1. *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 2. *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:

```rust
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 3. *Expliquer pourquoi cet API nous permettrait de partager des emprunts, et sous quelles circonstances.*

Exercice 4. *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`.

2 Atomiques

Fichier `src/main.rs`

La manière la plus simple de partager la même donnée mutable entre deux theads est d'utiliser des atomiques (module `std::sync::atomic`). Il est pertinent de regarder les fonctions disponibles sur les types `Atomicxxx`.

Exercice 5. *Compléter le code suivant, afin d'incrémenter le compteur dans les 2 threads en parallèle :*

```rust
let counter = /* */;
std::thread::scope(|scope| {
    scope.spawn(|| for _ in 0..100 {
        /* increment the counter by one */
    });
    scope.spawn(|| for _ in 0..100 {
        /* increment the counter by one */
    })
});
assert!(/* counter is equal to 200 */)
```

Remarque *: on utilisera toujours `Ordering::SeqCst` dans ce TD.*

Bien que les opérations fournies sur `Atomicxxx` soient *atomiques*, il est possible de commettre des erreurs de concurrence.

Exercice 6.

```
let counter = AtomicUsize::new(0);
std::thread::scope(|scope| {
    scope.spawn(|| for _ in 0..100 {
        let val = counter.load(SeqCst);
        std::thread::sleep(Duration::from_millis(1));
        counter.store(val + 1, SeqCst);
    });
    scope.spawn(|| for _ in 0..100 {
        let val = counter.load(SeqCst);
        std::thread::sleep(Duration::from_millis(1));
        counter.store(val + 1, SeqCst);
    });
});
println!("{}", counter.load(SeqCst));
```

Quels sont les résultats possibles du programme ci-dessus ?

3 Une 'base de données' concurrente

Fichier `src/database.rs`

Pour expliquer les difficultés de la concurrence nous allons implémenter une 'base de données' en utilisant une `HashMap` et des `RwLock`. Le type `RwLock` est une généralisation des mutex qui permet à plusieurs threads de lire la valeur *ou* à un thread d'écrire une valeur.

Remarques :

Les `RwLock` ont un API un peu particulier en Rust. Les fonctions principales `read` et `write` ont le type suivant

```
pub fn read(&self) -> LockResult<RwLockReadGuard<'_, T>>;
pub fn write(&self) -> LockResult<RwLockWriteGuard<'_, T>>;
```

Pour ce TP vous pouvez utiliser la fonction `unwrap` pour déplier le `LockResult` renvoyé. 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.

Ces types `RwLockReadGuard` et `RwLockWriteGuard` sont un exemple d'un pattern Rust appelé un 'drop guard'.

C'est un type qui agit *comme* une référence (mutable), mais qui a un comportement spécial : if *verrouille* le `RwLock` à sa création, et le *déverrouille* quand il sort du scope (quand il est `drop`).

Notons que les drop guards ont une différence fondamentale avec les références :

Exercice 7. *Le programme de droite est un équivalent, avec des* `RwLock`*, du programme de gauche :*

```rust
let mut x = 0;
let immut = &x;
println!("{immut}");
let mutable = &mut x;
*mutable += 1;
println!("{mutable}");
```

```rust
let x = RwLock::new(0);
let immut = x.read().unwrap();
println!("{immut}");
let mut mutable = x.write().unwrap();
*mutable += 1;
println!("{mutable}");
```

Mais le deuxième ne s'éxecute pas correctement : déterminer le début et la fin de chaque verouillage pour comprendre pourquoi.

Utiliser `std::mem::drop` *pour résoudre le problème.*

Pour implémenter notre base de données, nous utiliserons le type `Database` ci-dessous. On utilise deux niveaux de locks, un autour du `HashMap` pour contrôler les insertions et suppressions, 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 8. *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 9. *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 10. *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 11. *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 12. *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 13. *Vérifier que votre code (de la question précédente) réussisse le test* `deadlock`*. Le corriger sinon (on pourra utiliser* `RwLock::try_write`*).*

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