Travail pratique de programmation système avancée

Le pattern exécuteur - réacteur

1 Objectifs

- Familiarisation avec le pattern exécuteur - réacteur
- Exécution concurrente de plusieurs `Future`

2 Énoncé

Dans ce trasail pratique, inspirés par la séance sur les `exécuteur-réacteur` (voir `08_runtimes_wakers_reactor_executor.md`) nous aimerions implémenter une sorte de `join_all()` au sens de la séance sur les `coroutines` (voir `07_stackless_coroutines.md`) afin d'avoir une exécution potentiellement concurrente de nos `Future`.

2.1 Étapes: concurrence

1. Commencer par implémenter une machine d'états permettant de faire un unique `GET` sous la forme de `/{delay}/HelloWorld{i}`, avec `delay` un délai en milisecondes, et `{i}` le numéro de la requête. On aurait donc trois états: `Start`, `Wait`, `Resolved`
2. Implémenter une seconde machine d'états permettant de `spawn()` un nombre arbitraire de `Future` de la forme de la requête ci-dessus et qui aurait donc deux états `Start` et `Resovled`. Cette partie serait le `Future` parent des requêtes `GET`.
3. Appeler `block_on()` sur ce `Future` parent et observer le résultat.

2.2 Étapes: parallélisme

Effectuer la même chose, mais en mettant un exécuteur sur des threaeds séparés. Votre `main()` devra ressembler à quelque chose du genre:

```
let mut executor = runtime::init();
let mut handles = vec![];
for i in 1..NTHREADS {
  let name = format!("exec-{i}");
  let h = Builder::new().name(name).spawn(move || {
        // 1. Créer un nouvel exécuteur
        // 2. Bloquer sur async_main()
  }).unwrap();
```

```
    // 3. ajouter h à handles
}
// 4. bloquer dans le thread principal sur async_main également
// 5. join tous les handles
```

Mesurez les gains de performance!

2.2.1 Remarques

1. La fonction `async_main()` spawn 5 `Future` comme dans la partie 1.
2. Le `Builder` se trouve dans le dans `std::thread::Builder`