Genie Logiciel (GEN)

Labo 3 - Exécution en parallèle - Threads

Travail par groupe de 2-3 étudiants
Le travail sera évalué et valorisé pour 10% de la note de labo
*Délai de remise **: mercredi 25 mars** à minuit. Démonstration en séance le **jeudi 26 mars***

Introduction – Intégration continue avec Travis

Les threads sont un mécanisme de bas niveau mis en œuvre dans les systèmes d'exploitation pour permettre l'exécution en parallèle en mémoire partagée au sein d'une application. Consultez les transparents du cours **threads.pdf** sur ce chapitre.

Dans ce laboratoire, on réalisera deux exercices sur le threads en Java en vérifiant leur comportement avec JUnit et on en profitera pour faire exécuter ces tests dans un environnement d'intégration continue lié à Github: Travis. Consultez à ce sujet le guide « **Intégration continue.pdf** » diffusé avec l'énoncé de ce labo.

Votre projet IntelliJ contiendra donc 2 modules Maven, un pour l'exercice banque, un pour l'exercice lecteursredacteurs. Comme dans le labo2, associez-le à un repo GIT, mais cette fois un repo public sur GitHub. C'est nécessaire pour pouvoir utiliser Travis avec toutes les options.

Exercice Banque

Cet exercice a pour but de vous faire pratiquer les threads et les blocs synchronized.

Reprenez dans un package de votre module banque les classes Compte et Banque fournies dans **banque.zip**. Banque contient une liste de comptes pré-initialisés et une méthode de transfert entre 2 comptes qui maintient constant l'argent total en banque, ce qui est vérifiable par la méthode consistent().

Intégrez Travis sur votre repo et poussez ces classes dans la branche master. Passez ensuite dans une branche de développement **dev1**.

Ecrire une classe Transferts pour exécuter dans son propre thread N transferts entre comptes. Les comptes et les montants de chaque transfert sont tirés aléatoirement.

Dans une classe de test Junit, déclencher 1000 exécutions de la classe Transfert, chacune réalisant 1000 transferts, dans une banque de 10 comptes. Ne pas traiter l'accès concurrent aux comptes par les threads pour l'instant. Au contraire, on essaiera de produire un état inconsistant (méthode consistent) qui fera échouer le tests (quitte à ajouter des pauses dans le code).

Poussez cette branche de développement sur votre repo et constatez que les tests y échouent également.

Vérifiez que vous ne pouvez pas merger votre branche sur master si vous n'êtes pas connecté avec le compte propriétaire du repo. Vous pourriez le forcer avec le compte propriétaire mais ne le faites pas.

Laissez cette branche en l'état et passez dans une autre branche **dev2** dans laquelle vous corrigerez le problème au moyen de blocs synchronized. Essayez de ne verrouiller que les comptes nécessaires et non pas toute la banque. Des opérations sur des comptes différents doivent pouvoir être exécutées simultanément. Sans cela, vous perdez tous les avantages du parallélisme de l'exécution.

Le test JUnit doit alors passer. Vérifiez que vous pouvez cette fois merger la branche dev2 dans master.

Exercice lecteurs-redacteurs

Cet exercice a pour but de vous faire pratiquer le mécanisme wait notify.

Des lecteurs et des rédacteurs tentent d'accéder simultanément (chacun dans leur thread) au même document. Le principe est le suivant :

- Plusieurs lecteurs simultanés possibles
- Pas de lecteurs possibles lorsqu'un rédacteur accède au document
- Un seul rédacteur à la fois
- Priorité aux rédacteurs : un lecteur en attente ne peut pas accéder au document s'il existe un ou plusieurs rédacteurs en attente.
- Pas de priorité (passage aléatoire) entre les lecteurs en attente
- Pas de priorité (passage aléatoire) entre les rédacteurs en attente.

Si des rédacteurs arrivent alors que des lecteurs sont actifs et d'autres lecteurs sont en attente, les rédacteurs passeront juste après les lecteurs actifs, avant les lecteurs en attente.

Vous n'avez pas à programmer l'accès au document lui-même mais seulement les primitives de demande d'accès au document (méthodes startRead, startWrite) et sa libération (méthodes stopRead, stopWrite). Selon les règles évoquées ci-dessus, les primitives de demande d'accès peuvent déclencher une mise en attente et elles doivent donc être exécutée dans leur propre thread.

La classe LecteursRedacteursTest fournie dans **lecteursredacteurs.zip** contient un test commenté illustrant différentes situations de mise en attente.

Les lecteurs et les rédacteurs sont des objets des classe Lecteur et Redacteur et partagent un objet de la classe Controleur pour la gestion des accès. Développez ces classes en conformité avec le test fourni dans une branche **dev3** dont vous ferez un merge avec master sur votre repo.