Théorie et pratique de la concurrence – Master 1 Informatique
TP 2 : Programmation concurrente en Java (suite)

Exercice 1: *Le dîner des philosophes*

Le problème du diner des philosophes est un problème classique de partage de ressources en programmation concurrente. Ce problème peut être résumé de la façon suivante. Il y a n philosophes qui se trouvent autour d'une table. Chaque philosophe a devant lui une assiette de spaghetti. Directement à gauche de chaque assiette se trouve une baguette pour manger (il y a donc n baguettes). Un philosophe fait deux choses : penser et manger. Pour manger, il a besoin des deux baguettes qui sont à coté de son assiette. Chaque philosophe agit de la façon suivante : il pense, ensuite quand il a envie de manger il prend d'abord la baguette à sa gauche, *ensuite* la baguette à sa droite. Quand il termine de manger, il rend les deux baguettes et il pense, etc.

Le but de cet exercice est de modéliser ce problème du dîner des philosophes par un programme concurrent Java. Nous procéderons de deux façon différentes.

1. La première façon consistera à modéliser les baguettes par une classe `Baguette` avec deux méthodes `synchronized` pour prendre et relacher la baguette et un booléen spécifiant si la baguette est libre ou non. L'attente et le réveil des processus philosophes se fera grâce aux méthode `wait` et `notify` vues au TP précédent. Quant au comportement des philosophes, il sera modélisé par une classe `Philosophe` étendant la classe `Thread`.
   (a) Écrivez les classes `Baguette` et `Philosophe` décrites ci-dessus.
   (b) Testez votre programme avec 5 philosophes.
   (c) Essayez d'obtenir un deadlock en faisant "dormir" le processus caractérisant un philosophe à un certain moment.
   (d) Proposez une solution pour résoudre ce problème d'inter-blocage.
2. La seconde méthode consistera à utiliser des sémaphores pour représenter les fourchettes. Pour se faire, on utilisera la classe `Semaphore` du package Java `java.util.concurrent` (lisez la documentation avant de commencer à programmer).
   (a) Écrivez la classe `PhilosopheBis` utilisant des sémaphores pour les baguettes (n'oubliez pas que ces sémaphores seront partagés par différents philosophes).
   (b) Comme pour les questions précédentes, testez votre programme avec 5 philosophes et essayer d'obtenir un deadlock.

Exercice 2: *La piscine*

Une piscine dispose d'un certain nombre N_c de cabines et d'un certain nombre N_p de paniers. Un client qui se présente à l'entrée de la piscine effectue les étapes suivantes :

a. il doit attendre une clef de cabine,
b. il doit demander et obtenir un panier vide,
c. il occupe la cabine pour se changer,
d. il libère la cabine et la clef (et dépose son panier plein au vestiaire),
e. il va nager,
f. il cherche son panier plein au vestiaire et attend qu'une cabine se libère,
g. il prend une clef de cabine et occupe la cabine pour se changer, et
h. libère la cabine et rend la clef et le panier vide (et quitte la piscine).

Le but de cet exercice est de développer un programme concurrent simulant le comportement des clients de cette piscine.

1. Créer une classe `Client` étendant la classe `Thread` et dont la méthode `run` caractérise le comportement d'un client de la piscine. Les paniers et les cabines seront représentés par deux sémaphores `semCabine` et `semPanier` qui seront des champs de la classe `Client` (ces sémaphores seront partagés par tous les threads représentant les clients de la piscine).
2. Tester votre programme avec différents nombres de clients et différents nombres de cabines et de paniers (le nombre de cabines et de paniers étant déterminé par la valeur avec laquelle le sémaphore correspondant est initialisé). En particulier, essayer d'obtenir (en faisant dormir les threads à un certain endroit) un deadlock pour le cas où le nombre de cabines et plus petit que le nombre de paniers, lui même étant plus petit que le nombre de clients.
3. Dans le cas précis où il y a un deadlock, proposez une solution en modifiant le code des clients sans sans changer les nombres de clients, paniers ou cabines. **Indice :** Vous pouvez vous servir des méthodes proposées par la classe `Semaphore`.