TD de Compléments en Programmation Orientée Objet n° 6 : Multithreading : constructions de haut niveau

Exercice 1 : Le dîner des philosophes

On modélise la situation suivante : n philosophes sont assis autour d'une table, chacun ayant devant soi une assiette de spaghettis. Entre chaque assiette est posée une fourchette. Pour pouvoir manger, un philosophe à besoin d'utiliser les deux fourchettes les plus proches.

On précise que chaque philosophe essayera de manger à intervalles de temps aléatoires mais bornés, qu'il mange à chaque fois pendant une durée aussi aléatoire et bornée, et qu'une même fourchette ne peut être utilisée que par un seul philosophe en même temps.

1. Écrivez un programme Java qui simule cette situation.
2. Imaginez différentes variantes quant à la stratégie du philosophe (est-ce qu'il prend les deux fourchettes en même temps, peut-il libérer une fourchette sans avoir manger, etc.).
3. À chaque fois, décrivez quels scénarios indésirables pourraient se produire et essayez de trouver une solution (qui elle-même aura ses inconvénients...).

Exercice 2 : Arbres

Un arbre binaire est soit **null** (arbre vide), soit une racine avec deux sous arbres, c'est à dire un objet de la classe suivante :

```
1 class ArbreEntiers {
2     public int contenu;
3     public ArbreEntiers gauche;
4     public ArbreBinaire droite;
5 }
```

En utilisant des ForkJoinPool, implémentez les algorithmes suivants :

1. affecter zéro à chaque nœud.
2. calculer la branche la plus longue de l'arbre.

Exercice 3 : Compression en streaming

D'abord, quelques informations techniques/rappels :

— La méthode **static** ExecutorService newFixedThreadPool(**int** nThreads) de la classe Executors retourne un nouveau ExecutorService basé sur un pool de taille fixe de nThreads threads. Les tâches arrivent dans une unique file FIFO partagée, chaque thread prenant la première tâche en attente quand il se libère.

— La classe InputStream est munie d'une méthode **int** read(**byte**[] b, **int** off, **int** len), bloquante, qui permet de lire len octets et les copier dans le tableau b, à partir de la position off.

— La classe OutputStream est munie d'une méthode **void** write(**byte**[] b, **int** off, **int** len), qui permet d'écrire len octets pris depuis le tableau b, à partir de la position off.

On veut programmer une application de compression de flux audio à la volée. On suppose que le flux brut provient du microphone, via la carte son, sous la forme d'une série d'échantillons entiers (2 octets/échantillon), et qu'on veut écrire dans un fichier (.mp3, par exemple). Le flux d'entrée nous sera donné sous la forme d'un objet de type InputStream, et le flux de sortie sous la forme d'un objet de type OutputStream.

On dispose d'une classe Compresseur contenant la méthode **static byte[]** Compresseur.compresse(**byte[]** b) qui compresse le tableau d'octets b et retourne la version compressée. La documentation de cette méthode dit qu'elle est particulièrement efficace si on la fait travailler sur des blocs de 2^{16} octets. Elle retourne un tableau de taille variable.

1. Écrivez une classe dont le constructeur prend le flux d'entrée et le flux de sortie et qui possède une méthode qui compresse l'un vers l'autre en continu.
2. Écrivez une autre classe qui fait la même chose en répartissant le travail de façon efficace pour un système quadruple cœur à l'aide d'un pool de threads.