Les processus légers concurrents (threads)

Exercice 1 - Les Threads

Écrire un programme qui crée deux instances de la classe `Affiche1` et qui les démarre.

1 Écrire une classe `Affiche1` qui étend la classe `Thread`. Cette classe possède comme champ un entier identifiant chaque instance. Elle redéfinit la méthode `run()` de la classe `Thread` pour qu'elle affiche, dans une boucle de longueur n fixée, un message indiquant le numéro de l'instance courante et l'indice de la boucle.

2 Écrire une classe `Affiche2` fournissant les mêmes fonctionnalités que `Affiche1` mais sans hériter de `Thread`. Tester cette classe dans le main précédemment écrit.

Exercice 2 - Problèmes d'exclusion mutuelle

On cherche à étudier le comportement d'un objet modifier par plusieurs threads.

1 Écrire une classe `Point` avec deux champs entiers `x` et `y`, une méthode `void set(int x, int y)` (qui déplace le point aux coordonnées passées en arguments) et une méthode `toString()` qui affiche ce point au format `(x,y)`.

2 Écrire une classe `Imp` ayant un constructeur prenant en paramètre un `Point` et un entier `i`. La classe `Imp` possède une méthode `run()` qui démarre un processus léger qui appelle la méthode `set(i,i)` sur le point puis l'affiche, et ce dans une boucle infinie.

Créer deux instances de la classe `Imp` construites avec le même point `p` et des valeurs différentes (par exemple, `Imp(p,1)` pour l'un et `Imp(p,2)` pour l'autre). Démarrer les deux instances. Quels sont les différents affichages possibles pour le point ? Que faire ?

Exercice 3 - Producteur et consommateur

On désire développer une petite application simulant les comportements concurrents d'un ensemble de producteurs et de consommateurs de messages. D'un côté, les producteurs produisent des messages qu'ils stockent dans un buffer commun; de l'autre, les consommateurs récupèrent dans ce même buffer les messages (dans l'ordre où ils y ont été placés).

On doit respecter un certain nombre de contraintes et pouvoir paramétrer l'application:

1 un producteur ne peut produire un message que si le buffer n'est pas plein et, bien sûr, un consommateur ne peut lire un message que si le buffer n'est pas vide;

2 le `débit` de chaque producteur et de chaque consommateur est paramétrable à sa construction (il représente le temps d'attente entre deux productions ou deux consommations).

Écrire les classes `Producer` et `Consumer` permettant de mettre en oeuvre ces spécifications. On pourra implanter le buffer avec un `ArrayList` et chaque message produit doit être aisément identifiable. Chaque producteur et consommateur sera exécuté par un processus léger (`Thread`) différent, en concurrence avec les autres.

Exercice 4 - Sémaphore

Il n'existe pas de classe `Semaphore` en Java. Un sémaphore est un représentant d'un nombre de ressources fixé à l'avance. Implémenter la classe `Semaphore` sachant qu'elle possède les méthodes suivantes :

1. `retrieve()` qui essaye de recupérer une ressource s'il y en a une de disponible et qui sinon attend qu'une soit disponible.
2. `tryRetrieve()` qui marche comme `retrieve()` sauf renvoie `false` sans attendre que la ressource soit disponible.
3. `post()` qui redonne une ressource au sémaphore.
4. `count()` qui renvoie le nombre de resources restantes.

La classe `Semaphore` se contentera de simuler l'acquisition d'une resource en incrémentant un compteur interne.