CentraleSupélec

2ème année
Systèmes d'exploitation et réseaux

TL – Développement d'applications multi-threads en Java

Cette étude a deux objectifs : d'une part, d'illustrer la notion de thread (fil d'exécution), ainsi que les mécanismes de communication et de synchronisation entre threads ; d'autre part, de se familiariser avec les possibilités qu'offre un environnement de programmation et d'exécution particulier (ici Java) pour le développement d'applications multi-threads.

Cette étude est organisée de manière progressive. Elle se déroule sur deux séances. À la fin de la séance 1, vous devrez avoir traité au moins les parties 1 et 2. Chaque étape devra être validée par un enseignant.

Note : les références en gras sont des classes fournies par Java. Vous trouverez en bas du sujet les URL des plus importances pour cette étude. Les références en italique sont des constantes ou classes que vous devez définir ou implémenter.

1 Création de thread – synchronisation sur la fin d'un thread

Lorsqu'on lance un programme Java, cela crée un premier thread principal qui exécute le code de la méthode *main*.

Écrire un programme dont la méthode *main*, qui représente donc le thread principal, crée un thread secondaire. Le thread principal devra attendre explicitement la terminaison du thread secondaire avant de se terminer. Le thread principal devra afficher des messages lors de son lancement, après la création du thread secondaire, après le lancement puis après la terminaison de ce dernier. Le thread secondaire affichera un message juste après avoir démarré, et un message juste avant de se terminer.

Quand cette version simple fonctionnera, modifier le programme de sorte que le thread principal transmette deux arguments au thread secondaire lorsqu'il le crée : le nom du thread secondaire (une chaîne de caractères), et une durée en secondes (un entier). Le thread secondaire devra alors préfixer par son nom chacun des affichages qu'il produit, et se mettre en sommeil pour la durée indiquée avant de se terminer (voir **Thread.sleep**).

2 Implantation d'un protocole producteurs – consommateurs

2.1) Un producteur - un consommateur

On commence par traiter le cas d'un thread producteur échangeant des messages (ici, des nombres entiers) avec un thread consommateur. Ces échanges se feront à travers une boîte à lettres (en accès FIFO) de capacité *TAILLE_BOITE* messages.

Le producteur devra afficher un message à chaque fois qu'il aura déposé un élément dans la boîte à lettres, puis attendre un temps aléatoire (voir **Math.random**) avant de tenter de déposer la valeur suivante. On pourra convenir que le producteur dépose un nombre fixe de messages avant de se terminer, ou qu'il dépose une valeur conventionnelle (par exemple -1) pour indiquer la fin du flot de données. De même, le consommateur affichera un message à chaque fois qu'il aura récupéré un élément dans la boîte à lettres, et attendra un temps aléatoire avant de tenter de récupérer la valeur suivante. Sa terminaison se fera en fonction de la convention adoptée pour le producteur.

Réaliser quatre classes :

Main, contenant la fonction *main* (le « programme principal ») ;

Producteur, spécialisation de la classe **Thread**, contenant un constructeur et la méthode *run* ;

Consommateur, présentant les mêmes caractéristiques que la classe *Producteur*;

BoiteALettres, contenant un constructeur, ainsi que les méthodes *depose* et *retire*.

2.2) Plusieurs producteurs – plusieurs consommateurs

Modifier le programme précédent de manière à pouvoir traiter le cas de plusieurs producteurs et plusieurs consommateurs échangeant à travers plusieurs boîtes à lettres (un message n'est lu que par un seul consommateur). Ceci conduira notamment à pouvoir nommer les boîtes à lettres, à préciser, au moment de la création des threads producteurs et consommateurs, quelle boîte ils utilisent, et à adapter la politique de terminaison.

3 Un problème de synchronisation de threads : le salon de coiffure

On souhaite modéliser le comportement des clients et des coiffeurs dans un salon de coiffure. On considère que les clients et les coiffeurs ont leur propre flot de contrôle (ils s'exécutent dans des threads distincts). Il peut y avoir un nombre quelconque de threads, client ou coiffeur. Un thread principal crée des clients qui se présentent à l'entrée du salon de coiffure. Lorsqu'un client arrive, il repart sans se faire coiffer si le salon est plein, sinon, il s'installe dans un siège en salle d'attente et attend son tour. Lorsque le coiffeur a terminé la coupe d'un client, il indique au client suivant que c'est son tour et lui coupe les cheveux. Le client attend que le coiffeur ait terminé, puis s'en va.

3.1) Conseils sur l'ensemble des questions

- Identifiez les ressources partagées afin de déterminer de quels sémaphores vous avez besoin.
- Choisissez soigneusement l'objet auquel appartient un sémaphore.
- N'accédez pas directement aux sémaphores gérés par un objet, définissez plutôt une méthode dont le nom indique le sens de l'action effectuée. Par exemple, si un client possède un sémaphore *coupe* indiquant si sa coupe est terminée, il ne faut pas que le coiffeur accède directement à ce sémaphore pour faire un **release** dessus lorsqu'il a fini de coiffer un client. Il est préférable de doter les clients d'une méthode *finDeCoupe(),* qui sera appelée par le coiffeur, et qui en interne effectuera *coupe.release()*.
- Travaillez avec des modèles simples et de petite taille. Par exemple, 2 coiffeurs, 4 sièges d'attente et 10 clients suffisent largement lorsqu'on veut étudier l'imbrication des différents comportements. Faites varier le rapport entre le temps entre l'arrivée de deux clients et la durée d'une coupe afin d'explorer plus de possibilités.



3.2) Premier modèle

En vous inspirant de l'exemple du producteur et du consommateur, modélisez le salon de coiffure en le considérant comme une file d'attente pour les clients. Votre programme principal devra créer un salon de coiffure disposant d'un certain nombre de sièges d'attente, un coiffeur travaillant dans ce salon, puis un certain nombre de clients en attendant un temps aléatoire entre chaque création de client. Le coiffeur prendra un temps aléatoire pour couper les cheveux d'un client.

Dans ce modèle, le programme principal est le producteur (même si ce sont les clients qui se placent dans la file d'attente lorsqu'il y a des sièges disponibles), et le coiffeur est le consommateur (c'est lui qui retire les clients de la file d'attente). Les principales différences par rapport au producteur-consommateur sont :

- un client ne reste jamais bloqué en attente d'un siège car s'il n'y a pas de place, il repart sans se faire couper les cheveux (voir la méthode **tryAcquire** de **Semaphore**) ;
- ne fois assis dans un siège, le client attend que le coiffeur s'occupe de lui, puis attend que le coiffeur ait fini de lui couper les cheveux.

3.3) Un modèle plus complet

Dans la pratique, quand c'est le tour d'un client, ce dernier se lève de son siège d'attente et vient s'installer dans le fauteuil de travail du coiffeur. Modélisez l'acquisition et la libération de cette nouvelle ressource par le client. Le coiffeur devra vérifier que son client est bien installé avant de commencer à le coiffer.

Vous devez pouvoir facilement adapter votre code pour que plusieurs coiffeurs, ayant chacun son propre fauteuil de travail, travaillent dans le même salon et puisent donc des clients dans la même file d'attente.

3.4) Pour finir

En réalité, quand aucun client n'est présent, le coiffeur ne reste pas figé à attendre. Modélisez le fait que si la file d'attente est vide, le coiffeur s'installe dans son fauteuil et s'endort. Il se réveillera lorsque de nouveaux clients arriveront. Le premier de ces clients devra attendre que le coiffeur ait libéré le fauteuil pour s'y installer.

Références Java

- http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html
- http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html
- http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html