AMAURIN Alexandre
BECHER Hervé
RICM4

Décembre 2016

Compte-rendu projet système
Programmation concurrente

Quelques mots

Ce projet a été réalisé en binôme en utilisant la plateforme GitHub ainsi que l'environnement de développement Intellij IDEA. Le projet, une fois terminé, a été exporté au format Eclipse.
Les sources du projet sont disponibles ici.

Nous avons fait le choix de rendre chaque version indépendante, dans le sens où v(x + 1) n'hérite pas de v(x), afin de simplifier la lecture du code quitte à rajouter énormément de redondance.

I. Objectif n°1

Le premier objectif consiste en l'implémentation de la solution directe. Celle-ci a été faite exactement comme vu en TD avec le tableau de gardes-actions suivant :

Classe ProdCons	**Garde**	**Action**
Tampon.get(...)	`enAttente() > 0 || producteurs == 0`	<retirer le message>
Tampon.put(...)	`enAttente() < taille()`	<insérer le message>

A noter que l'on vérifie le nombre de producteurs dans la garde de `get(...)`, pour que les consommateurs puissent s'arrêter après que tous les producteurs aient fini. En effet, un message de type `MessageEnd` (qui implémente `Message`), est renvoyé dans le cas où il n'y a plus de producteurs actifs, ce qui cause l'arrêt du consommateur. Dans le cas contraire, les consommateurs retirent des messages de type `MessageX`.

Un `MessageX` est caractérisé par :

- le producteur qui l'a créé
- un identifiant unique propre au producteur (message n°X de ce producteur)
- une chaîne de caractères quelconque (ici utilisée pour stocker le moment de création du message, renvoyé par `System.currentTimeMillis()`)

Pour implémenter l'objectif, nous avons mis environ deux heures, le temps de bien comprendre l'énoncé et d'avoir la hiérarchie de base de l'application ainsi que le Git prêt à l'emploi.

Plusieurs tests ont été effectué, jouant sur le nombre de producteurs et consommateurs (1 à plusieurs de chaque), la taille du tampon (1 à très grand) et les temps de consommation (vérification des interblocages).

II. Objectif n°2

Dans cet objectif, il fallait mettre au point une solution utilisant nos propres sémaphores. La classe `Semaphore` a été créée pour cela dans le paquetage `v2`.

Cette classe reprend la spécification des sémaphores vue en cours et est utilisée comme vu dans les TDs. Cependant, les méthodes `P()` et `V()` ont été appelées comme leurs homologues Java, pour faciliter les tests.

Cet objectif a été rempli en 30 minutes environ.

III. Objectif n°3

Ici, nous avons ajouté les appels à l'observateur aux endroits adéquats de l'application. Une dizaine de minutes ont été nécessaires pour les vérifications.

IV. Objectif n°4

Jusqu'à présent, les classes `Producteur` et `Consommateur` n'avaient pas été modifiées (hormis pour ajouter les vérifications de l'observateur).

Pour satisfaire l'objectif, un nouveau type de message a été créé : `MessageTTL` (un message avec un certain nombre d'exemplaires, le nom est une analogie au temps de vie des paquets Ping). Cette classe hérite de `MessageX` et dispose du nombre d'exemplaires restant du message à retirer avant sa destruction. Lorsqu'un producteur envoie un message, celui-ci se bloque sur un sémaphore qui lui est propre. Il se réveille lorsque le dernier exemplaire de son message est consommé.

Contrairement aux précédents objectifs, celui-ci a pris plus de cinq jours pour être complété. Mise à part la mauvaise compréhension de l'objectif au départ (vite réglée), deux fautes non vues étaient responsables pour cette prise de temps (un `notifyAll()` non repéré à la place d'un `notEmpty.release()` et une libération prématurée des producteurs suite à une absence de vérification sur le sémaphore `notFull` après retrait d'un exemplaire de message non terminal).

V. Objectif n°5

Cet objectif a été rempli en relativement peu de temps, une dizaine de minutes au plus. Le `Lock` partagé entre les `Conditions` joue le rôle de mutex, et `notEmpty` et `notFull` sont les régulateurs de passage (comme pour les sémaphores de la version 3).

VI. Objectif n°6

Cet objectif n'a pas été rempli dans son intégralité (moins d'une soirée pour le faire), néanmoins quelques vérifications de base ont été implémentées.

La classe `Vérificateur` contient une file qui garde en mémoire l'ordre d'arrivée des messages.
Lors d'un retrait, on compare le message retiré au premier message inséré dans la file ; si les deux sont différents, alors la propriété fifo n'est pas respectée et une `VerifException` est lâchée.
De même, lorsque tous les producteurs et consommateurs ont fini leurs activités, le vérificateur vérifie que tous les producteurs ont produit leur quota de messages, et que le tampon est bien vide, sinon une exception est lâchée.

VII. En conclusion

Au total, nous avons passé un peu plus de deux semaines sur ce projet, et complété la majorité des objectifs.