MARCHAND Charles
PELLICER Marion

14/12/2016
RICM4

SYSTEME D'EXPLOITATION - RAPPORT PROJET : PROBLEME PRODUCTEUR/CONSOMMATEUR

Introduction :

Ce projet a été réalisé par MARCHAND Charles et PELLICER Marion. Le temps effectif utilisé pour l'élaboration de ce projet est d'environ 32h. Nous avons atteints les objectifs 1 à 5.

Objectif 1 :

L'objectif 1 a nécessité la création de 5 nouvelles classes qui serons réutilisées tout au long du projet et qui ne subiront des modifications plus ou moins légères afin de répondre aux différents objectifs demandés. Nous avons donc implémenté les classes *Consommateur*, *Producteur*, *ProdCons* (représentatif du tampon), *MessageX* et *TestProdCons* (contenant la méthode main).

Ce premier objectif avait pour but d'implémenter le problème du producteur/consommateur à l'aide de la solution directe des *wait()* et *notify()* fournit par java. Nous n'avons rencontré aucun problème pour cet objectif d'autant plus que nous avions déjà vu son fonctionnement en cours ainsi qu'en travaux dirigés.

Objectif 2 :

L'objectif 2 consistait en la reprise de l'objectif 1 à l'aide de nos propres sémaphores. Nous avons donc créer une nouvelle classe *MonSemaphore* représentant nos propres sémaphores et pour lesquels les méthodes *P()* (et *V()*) correspondant aux méthodes *acquire()* (resp. *release()*) déjà existentent dans la classe *Semaphore* fournit par java.

Encore une fois c'est une méthode qui a été vue à plusieurs reprises. Le fait d'implémenter nos propres sémaphores nous a permis de constater que nous avions bien compris le fonctionnement d'un sémaphore.

Objectif 3 :

Pas grand-chose à dire là dessus.

Objectif 4 :

Voilà l'objectif qui nous a posé le plus de problèmes. Toutes les heures passées sur le projet sont dûes à cet objectif. Nous n'avons pas eu le problème de la compréhension du sujet que certains ont pu avoir. Cependant en discutant un peu avec les autres groupes travaillant sur le projet, nous avons constaté qu'aucun des groupes auxquels nous avions parlé n'avait implémenté la solution comme nous l'avons fait.

Pour répondre à la problématique de l'objectif nous avons opté pour l'ajout d'un sémaphore de 1 propre à chaque *consommateur* et à chaque *producteur*. L'intérêt d'un tel sémaphore est de bloquer l'activité de l'acteur auxquel il appartient tant que tous les exemplaires d'un message n'a pas été lu. Plus précisément, celui-ci bloque un producteur tant que le dernier message posé n'a pas été lu autant de fois qu'il le devait et bloque un consommateur tant que le message accessible dans le buffer a déjà été traité par lui.

Pour relacher ce sémaphore au bon moment il nous a fallu fournir à notre classe *ProdCons*, une liste d'acteurs qui va stocker tous les acteurs qui sont actuellement en attente d'un release de sémaphore.

Enfin, chaque *MessageX* possède un entier *nombreDeRepetition* qui indique combien de fois celui-ci doit être lu avant d'être retiré du buffer.

Objectif 5 :

Cette objectif demande l'utilisation de la bibliothèque *java.util.concurrent* qui implémente les « Condition ». Nous avons donc repris l'objectif n°3 en remplaçant l'utilisation de nos sémaphores par celle des *locks* et des *conditions*. Nous avons vu en TD différentes méthodes qui implémentaient les sémaphores pour simuler une politique de priorité, mais aucun exemple avec des verrous et des conditions, ceci étant dit on retrouve bien la même idée derrière tout ça. Nous nous sommes donc renseignés sur la bibliothèque *java.util.concurrent* et nous sommes aidés des exemples fournis pour implémenter notre solution.

Objectif 6 :

Dans cet objectif, il nous faut spécifier et réaliser notre propre mécanisme d'observation montrant le respect des propriétés du protocole pour l'objectif n°3. Pour cela, nous avons créé 7 exceptions qui sont ici pour controler que nos conditions sont bien respectées. Dans le cas où tout ne serait pas bien respecté, l'exception associée à l'erreur sera appelé et l'utilisateur verra alors apparaître cette exception.