POLYTECH®
GRENOBLE

SYSTEME D'EXPLOITATION

PROGRAMMATION CONCURRENTE

GERRY Quentin & RACHEX Coralie

-

RICM4

Objectif général

Le but de ce TP est de comprendre les mécanismes de la programmation concurrente. Pour cela, nous intéresserons au modèle des "Producteurs / Consommateurs". Ce modèle est un modèle classique de synchronisation entre les threads producteurs qui déposent des messages dans un tampon de taille bornée et les threads consommateurs qui les retirent pour les consommer. En considérant chaque case du tampon comme une ressource, le but est de synchroniser les threads de sorte à ne pas écraser de l'information dans une case déjà remplie, ou ne pas retirer d'une case de l'information inexistante. De plus, on souhaite que les informations produites soient consommées dans le même ordre. Pour ce faire, le tampon est géré de façon circulaire.

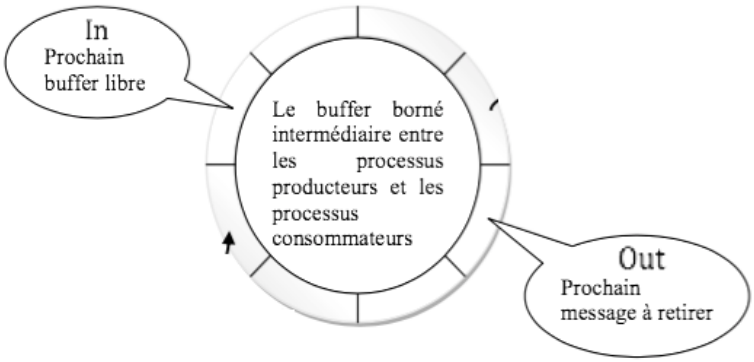
Le problème de producteurs/consommateurs est donc à la fois un problème d'exclusion mutuelle et un problème de synchronisation :
- une même ressource ne doit pas être utilisée par plusieurs threads en même temps
- un producteur ne doit pas ajouter des données sur une file pleine
- un consommateur ne doit pas retirer des données d'une file vide

I. Objectif 1

1. Réalisation de l'objectif 1

Ce premier objectif a été long à réaliser (~8h) car nous avons dût réfléchir à l'implémentation de l'ensemble des classes. Dans ce premier objectif, nous avons utilisé la méthode directe "Garde/Action" :

Opération	Garde	Action
put(producteur, message)	! isPlein()	nbMsg++
get(consommateur)	! isVide()	nbMsg--



a- Classe Producteurs

Il s'agit de la classe des threads producteurs, qui hérite donc de la classe Thread. Comme la méthode run() de la classe Thread ne fait rien, il faut la surcharger. Cette méthode run() permet de produire les messages puis doit invoquer la méthode put() de la classe ProdCons, pour les déposés sur le tampon.

b- Classe Consommateurs

Il s'agit de la classe des threads consommateurs, qui hérite donc de la classe Thread. Comme la méthode run() de la classe Thread ne fait rien, il faut la surcharger. Cette méthode run()doit invoquer la méthode put() de la classe ProdCons, pour retirer les messages du tampon, et elle peut ensuite les traiter.

c- Classe ProdCons

Cette classe représente le tampon et les méthodes qui lui sont associées. Le tampon (ou buffer) est représenté comme un tableau de message, auquel les consommateurs accèdent grâce à l'indice out et les producteurs grâce à l'indice in. Autrement dit, les producteurs et les consommateurs ne s'intéressent pas à la même variable, il n'y a donc pas de conflit entre un producteur et un consommateur. En revanche, il peut y avoir des conflits entre les consommateurs ou entre les producteurs, puisqu'ils s'intéressent à la même variable. C'est pour cela que les méthodes put() et get() sont "synchronized" et que nous utilisons la méthode de garde/action. De cette façon ces méthodes seront considérées comme exclusives et non interruptible (une seule instance de la méthode pourra être exécutée à tout instant).

Autrement dit, les méthodes Put et Get sont synchronisées et encadrées par des wait et notifyAll avec un while devant le wait qui contient la garde en question (condition qui doit être vérifiée pour permettre l'accès à la ressource). Le notifyAll permet quant à lui de réveiller tous les threads bloqués de sorte que le programme puisse continuer.

d- Classe MessageX

Cette classe représente les messages déposés sur le buffer. Dans la première version ces messages contiennent uniquement l'indice du producteur ayant produit le message ainsi que le numéro du message.

e- Classe TestProdCons

C'est la classe qui contient le main permettant d'exécuter le programme. La méthode main invoque la méthode start(), qui va a son tour invoquer la méthode run(). Cette méthode run() permet d'initialiser le programme en parsant le fichier XML, puis en créant les threads producteurs et consommateurs. Ces différents threads sont exécutés grâce à la méthode start(), qui va invoquer la méthode run() des classes Consommateur et Producteurs.

2. Tests de l'objectif 1

Plusieurs fichiers XML sont disponibles dans le dossier option fournit dans notre archive. Le but de ces tests est explicité en commentaire au début de chacun d'eux, et est repris à la fin de notre rapport. Ils nous ont permis de tester différents cas :

- Vérifier que les producteurs sont bien bloqués lorsque le buffer est plein.
- Vérifier que l'on ne dépasse pas la taille du buffer
- Vérifier que les producteurs déposent tous leurs messages
- Vérifier que les consommateurs récupèrent les messages dans l'ordre déposé, pour que les producteurs puissent de nouveau déposer leur message.
- Vérifier la bonne terminaison du programme.
- Vérifier que l'exécution se déroule bien lorsque les délais de consommation ou de production sont élevés.

II. Objectif 2

Cet objectif a été plus rapide à mettre en place (~2h) puisqu'il s'agissait essentiellement de modifier la classe ProdCons afin de remplacer le système de Garde/Action par des sémaphores.

Comme nous l'avions déjà dit, les producteurs et les consommateurs ne s'intéressent pas à la même variable (in/out). En revanche, la variable "nbCasePleine" est modifiée par les deux et est utilisée comme garde par les deux. Une solution est de créer une variable "nbCaseVide" pour que les Consommateurs ne travaillent pas sur la même variable que les Producteurs.

Nous avons donc créé une nouvelle classe sémaphore, contenant deux méthodes synchronisées : p() pour faire attendre les Threads et v() pour réveiller les Threads. Cette classe contient l'attribut "residu" qui symbolise "nbCasePleine" ou "nbCaseVide" selon l'initialisation qui sera faite lors de l'instanciation. Autrement dit, dans ProdCons, nous avons deux instances de la classe Sémaphore, une instancié avec residu = 0 pour bloquer les consommateurs qui doivent attendre que le buffer contienne un message, et une autre instanciée avec residu = tailleBuffer pour les producteurs qui ne peuvent déposer des messages que tant que le buffer n'est pas plein.

Nous avons donc un système avec deux sémaphores et deux blocs synchronized :
- fileCons : Bloque les Consommateurs lorsque le buffer est vide
- FileProd : Bloque les Producteurs lorsque le buffer est plein
- Blocs synchronized : Permet l'accès unique à buffer[in] et à buffer[out]

Il faut ici faire attention à enlever le synchronized des méthodes put() et get() pour éviter l'imbrication de méthodes synchronisées.

III. Objectif 3

Cet objectif a également été rapide à mettre en œuvre (~2h), puisqu'il s'agit de placer les fonctions de la classe observateur aux bons endroits dans nos classes.

Dans TestProdCons, on appelle la méthode d'initialisation de l'observateur ainsi que les méthodes qui permettent de notifier à l'observateur la création de nouveau threads consommateur et producteur.

Les méthodes de production et de consommation de messages sont insérées dans les fonctions run() des Producteurs et des Consommateurs (respectivement). Alors que les fonctions de dépôt et de retrait des messages sont insérées dans les méthodes put() et get() de la classe ProdCons, puisque celles-ci ont un accès concurrent au buffer.

IV. Objectif 4

Cet objectif a été long à mettre en œuvre (~8h), car nous avons rencontré des problèmes difficiles à déboguer dans la classe ProdCons.

Le but de l'objectif est de modifier le code de sorte qu'un même message soit déposé en n exemplaires et qu'il soit donc lu n fois avant de disparaître du buffer. Tant que le message n'a pas disparu du buffer, le Producteur ayant déposé le message ne peut pas déposer d'autre message.

Pour pouvoir réaliser cela, nous avons modifié la classe MessageX de façon à ajouter la gestion du nombre d'exemplaires du message. Nous avons donc une méthode qui permet de consommer un exemplaire du message et une méthode permettant de vérifier si tous les exemplaires ont été consommés. Ces deux méthodes sont utilisées dans la méthode get(), pour ne retirer à chaque fois qu'un seul exemplaire du message, puis pour effectuer un test afin de savoir si l'on retire définitivement le message du buffer ou non.

Nous avons modifié la classe Semaphore de façon à pouvoir ajouter n résidus dans la fonction v(). En effet, lorsque les Producteurs déposent un message sur le tampon, il faut qu'ils puissent notifier aux Consommateurs que n exemplaires d'un message ont été ajoutés sur le buffer et attend donc d'être lu. Ainsi, les Consommateurs n'accèdent au buffer que si des exemplaires sont disponibles sur le buffer (on s'intéresse au nombre d'exemplaires et non plus au nombre de message).

Au final, nous utilisons trois sémaphores et des blocs synchronized:
- **FileProd** : Bloque les Producteurs lorsque le buffer est plein
- **ExemplaireCons** : Bloque les Consommateurs lorsqu'il n'y a plus d'exemplaires d'aucun message à lire sur le buffer
- **ExemplaireProd** : Bloque le Producteur tant que tous les exemplaires du message qu'il a déposé n'ont pas été lus

V. Objectif 5

Cet objectif a été rapide à mettre en œuvre (~2h), puisque nous avons réutilisé notre code en replaçant seulement les sémaphores par des Lock et Condition de la bibliothèque java.util.concurrent.

Les conditions fournissent un moyen de suspendre l'exécution d'un thread si la condition est vraie et ce jusqu'à ce qu'un autre thread lui notifie que l'état de la condition est devenue fausse. Puisque l'accès à l'état de la condition est partagé dans différents threads, il doit être protégé par un Lock.

VI. Objectif 6

L'objectif 6 a été relativement long ~20h. Un première version à été faite mais la javadoc n'avait pas été respecté (nous ne l'avions pas vue) et nous avons donc du refaire une nouvelle version pour respecter la javadoc.

Nous avons donc crée une classe Observator ainsi qu'une classe Contrôleur.

Class Observator :
Permet de faire le lien entre le la simulation et notre contrôleur.

Class Controleur :
Le contrôleur contient ses propres exception :

Nom	Utilité
ExceptionSimulation	Exception global (classe mère)
ExceptionProduction	Exception lors de la production d'un msg
ExceptionConsommation	Exception lors de la consommation d'un msg
ExceptionDepot	Exception lors du dépot d'un msg
ExceptionRetrait	Exception lors du retrait d'un msg
ExceptionSizeBuffer	Exception concernant la taille du buffer

Le contrôleur est composé de diverse fonction pour permettre un monitoring avancé de la simulation :

Nom	Utilité
consommationMessage	Évènement correspondant à la consommation d'un message : on ajoute le message consommé au consommateur
depotMessage	Évènement correspondant au dépôt d'un message dans le tampon : On rajoute dans la file le message déposer et on l'enlève du producteur
init	initialisation de l'observateur : Initialise la liste des producteurs pour vérifier la production
initProd	Initialise le nombre de message que le producteur doit produire
newConsommateur	Évènement correspondant à la création d'un nouveau consommateur : On rajoute le consommateur
newProducteur	Évènement correspondant à la création d'un nouveau producteur : On rajoute le producteur
productionMessage	Évènement correspondant à la production d'un message : On rajoute le message produit au producteur
retraitMessage	Évènement correspondant au retrait d'un message du tampon: On enlève le message de la file
displayConsommateur	On affiche les consommateurs et les messages qu'ils ont consommé
isFamine	On détecte la famine et on affiche les consommateur en famine si display = true
isCoherent	On définie sur la simulation est cohérente et on affiche les problèmes de cohérence si display = true

VII. Jeu de test

Afin de tester nos différentes versions, nous avons réalisé un jeu de test. Ce même jeu de test a été utilisé pour vérifier la fiabilité de notre contrôleur :

Nom	Fonction
optionsDevia1	Cas où le temps moyen de production peut valoir 0 : *Levé d'exception*
optionsDevia2	Cas où le temps moyen de Consommation peut valoir 0 : *Levé d'exception*
optionsDevia3	Cas où temps de production est important
optionsDevia4	Cas où le temps de consommation est important
optionExemp	Cas où le nombre d'exemplaires peut être nul
optionsMultiBuff	Cas où il existe plusieurs buffers et peu de consommateurs : permets de vérifier que lorsque les threads producteurs sont morts, les threads consommateurs continuent de vider le buffer avant de mourir.
optionsNoBuffer	Cas où il n'existe pas de buffer : *Levé d'exception*
optionsTailleBuff	Cas où la taille du buffer est de 1 : test du non-dépassement de la taille du buffer
optionsNoCons	Cas où il n'existe pas de consommateur : permets de vérifier que les producteurs sont bien bloqués lorsque le buffer est plein : *Levé d'exception*
optionsNoProd	Cas où il n'existe pas de producteur : permets de vérifier que les threads consommateurs sont bien tués quand il n'y a pas de message sur le buffer et que les producteurs sont morts : *Levé d'exception*
optionsOverload	Stress tests