UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

université PARIS-SACLAY

Programmation multitâches en mémoire partagée

Rapport – Programmation répartie

Maxime VINCENT – INFO2-FI-B

Rapport des trois premières séances de TP sur la programmation multitâches en mémoire partagée en Java.

Table des matières

I. Introduction

Définition :

Application répartie :

Une application répartie peut permettre l'interaction de plusieurs applications séparées.

Système répartie :

"Un système réparti est un ensemble de machines autonomes connectées par un réseau et équipées d'un logiciel dédié à la coordination des activités du système ainsi qu'au partage de ses ressources." [Coulouris et al, 1994]

Contexte :

Dans ce rapport nous allons voir l'utilisation de la programmation répartie avec des Threads[1] en java.

Java dispose d'un mécanisme de "processus légers" (threads) qui s'exécutent en parallèle au sein d'une même Java Virtual Machine. Parfois sur plusieurs cœurs ou processeurs selon le type de support physique utiliser.

[1] Processus léger qui possède ça propre mémoire.

II. TP1 : Introduction à l'utilisation des Threads en Java

a. Les constructeurs de la classe Thread

Tableau des constructeurs de la classe Thread :

Thread ()	Créer un nouveau thread
Thread (Objet Runnable)	Créer un nouveau thread en lui précisent le code à exécuter
Thread(String name)	Créer un nouveau thread en lui précisant son nom
Thread(Objet Runnable, String name)	Créer un nouveau thread en lui précisent le code à exécuter et son nom

Objet Runnable : Objet qui implémente l'interface Runnable et qui possède une méthode run().

b. Méthodes de la classe Thread

Tableau des méthodes de la classe Thread :

start ()	Appel la méthode run() du thread
suspend ()	Permet de suspendre l'exécution d'un thread
resume ()	Permet de reprendre l'exécution d'un thread mis en pause

c. Utilisation de la classe Thread

Nous pouvons créer un objet d'une classe implémentant l'interface Runnable et composé d'une méthode run().

Par la suite ont créé un objet de la classe Thread en lui passant dans le constructeur l'objet Runnable. Puis on appelle la méthode start() de l'objet Thread, cela va déclencher la méthode run().

On peut mettre en pause le thread avec la méthode suspend() et le reprendre avec la méthode resume().

III. TP2 : Thread, Exclusion mutuelle avec synchronized, Sémaphores

a. Exclusion mutuelle avec synchronized

On peut créer une classe qui hérite de la classe Thread, cette classe aura obligatoirement une méthode run() pour préciser le code à exécuter au thread (comme les classes qui implémentent runnable).

Nous pouvons synchroniser les threads avec le bloc suivant :

synchronized(Objet *static* obj){ "section critique" }

Synchronized prend en paramètre un objet *static* pour que tous les threads puissent le connaitre.

Une section critique est une partie du code ou s'exécute qu'un thread à la fois. Une section critique est utilisée lorsque plusieurs threads accèdent à une même ressource.

b. Sémaphores

Dans une classe sémaphore on crée 2 méthodes et 1 attribut. (Voir ci-dessous)

Tableau des attributs de la classe Semaphore :

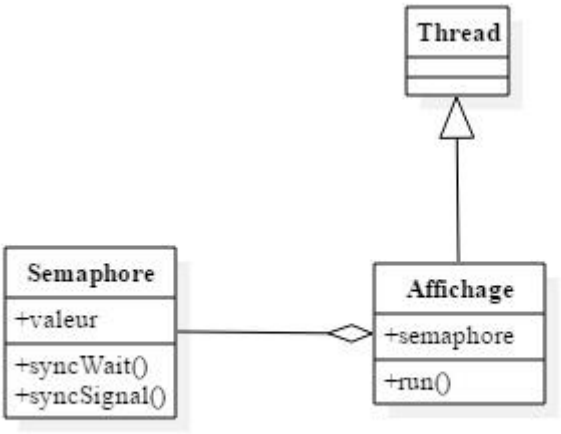
valeur	Entier initialiser à 1 (permet de faire le verrou des threads) [0 à -infini]

Tableau des méthodes de la classe Semaphore :

syncWait()	Tant que la valeur est inférieur à 0 appel la méthode wait() puis décrémente la valeur.
synSignal()	Incrémente la valeur si elle est supérieur à 0 envoie un signal au thread bloquer avec la méthode *notifyAll()*

Pour synchroniser des threads on crée un objet de classe sémaphore en *static*. Puis on encadre la section critique avec les méthodes syncWait() et synSignal() du sémaphore.

Voir ci-dessous : diagramme de classe du modèle sémaphore



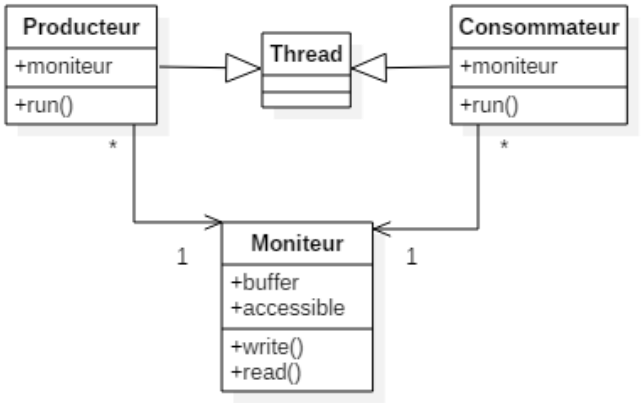
IV. TP3 : Thread, Moniteur, Modèle producteur consommateur

a. Introduction modèle producteur consommateur

Un moniteur est un objet de synchronisation qui permet l'exclusion mutuelle et d'attendre avec ma méthode *wait()* qu'une condition soit validée.

Le moniteur à des méthodes qui sont le point d'entrée des processus légers (Thread). Il possède un seul thread actif à la fois, les autres sont bloqués. Cela permet de réunir les opérations de synchronisation au même endroit.

Voir ci-dessous : diagramme de classe du modèle producteur consommateur simple :



b. Exemple Boite à lettre

Dans la partie I1 du TP 3 :

Le moniteur est la boite à lettre « **classe BAL »**, sa condition est que la boite à lettre ne peut pas contenir plus d'une lettre,

Tableau des attributs de la classe BAL :

Buffer	String contenant la lettre ou rien
Plein	Booléen qui passe a vrai s'il y a une lettre dans le buffer faux sinon

Tableau des méthodes de la classe BAL :

retirer()	Méthode « *synchronised* » elle attend tant que le « *buffer* » soit vide (« *wait()* ») puis récupère le contenu du buffer. Passe la variable *plein* à faux puis préviens tous les threads que le travail est terminé (« *notifyAll()* »).
deposer(String lettre)	Méthode « *synchronised* » elle attend tant que le « *buffer* » soit remplie (*« wait() »*) puis dépose le paramètre « *lettre* » dans le buffer. Passe la variable *plein* à vrai puis préviens tous les threads que le travail est terminé (« *notifyAll()* »).

Les classes héritant de Threads :

Classe **Producteur –** est une classe héritant de Thread. Elle prend en paramètre une BAL et un String elle possède une méthode *run()* qui appel la méthode *deposer()* de l'objet de la classe *BAL.*

Classe **Consommateur -** est une classe héritant de Thread. Elle prend en paramètre une BAL elle possède une méthode *run()* qui appel la méthode *retirer()* de l'objet de la classe *BAL.*

Pour simuler la boite à lettre, dans la méthode main on instancie un objet de la classe **BAL** et des objets des class **producteur** et **Consommateur**. Puis on appelle la méthode *start*() de producteur et de consommateur. Pour terminer on interrompt l'exécution des processus légers avec les méthodes interrupt() et join().

Dans la partie I2 :

On ajoute un **Scanner** pour récupérer une entrée de l'utilisateur dans la méthode *run()* de la classe **Producteur**, puis on ajoute une variable booléenne « *fin*i » dans la classe **BAL** qui est égale a *false* tant que les producteurs continu d'entrer des lettres. On ajoute une condition dans la méthode déposer de la classe BAL qui passe la variable « *fini* » a *vrai* si un producteur entre 'q' ou 'Q'.

On peut observer que tant que le producteur n'a rien saisi donc tant qu'il n'a pas déposé une lettre dans la Boite à lettre, le consommateur ne lit rien. C'est grâce à l'objet de synchronisation BAL que l'on a créé.

V. Annexes

Lien vers Git : https://github.com/EmixVCT/PROG-REPARTIE