Construction de Solutions Directes

→Application au cas de l'allocation de ressources

Cours d'Applications Concurrentes, *Polytech-INFO4*

F. Boyer, Université Grenoble Alpes, année 2022-2023

1. Introduction

Nous allons dans ce TD/TP étudier une méthodologie aidant à produire une solution correcte, à base de moniteurs, à un problème de programmation concurrente. A titre d'exemple, nous allons considérer le problème général de l'allocation de ressources, qui consiste à programmer une classe (que l'on appellera *Allocator*) permettant à des threads de prendre et de libérer des ressources indifférenciées.

L'interface fournie par la classe *Allocator* est la suivante. Le constructeur de la classe permet d'initialiser le nombre de ressources disponibles initialement.

```
class Allocator{
        public Allocator (int nressources) ;
        public void get() ;
        public void put() ;
}
```

2. Démarche

La démarche suivie va consister en deux étapes:

1. Modélisation du problème en termes de variables
2. Définition du tableau de Gardes/Actions (GA)
3. Dérivation (automatique) de la solution directe à partir du tableau GA

L'étape 1 définit les variables dont nous avons besoin pour mettre en œuvre les méthodes attendues. Dans le cas présent, nous avons besoin d'une variable (int *nfree*) qui indique le nombre de ressources libres à l'instant courant.

L'étape 2 définit le tableau GA comme suit, avec 1 ligne par méthode. Le champ *Garde* exprime la condition de réalisation de la méthode (condition pour que la méthode puisse être exécutée totalement). Cette condition porte sur des variables définies à l'étape 1. Le champ *pre-action* exprime les modifications qui doivent être apportées à ces variables avant

d'évaluer la garde. Le champ *post-action* exprime les modifications qui doivent être apportées à ces variables une fois la garde satisfaite.

Méthode	**Pre-action**	**Garde**	Post-**action**
get()	-	nfree > 0	nfree-=1
put()	-	-	nfree+=1

L'étape 3 produit une solution (appelée solution directe) de manière automatique à partir du tableau GA. Ainsi, les méthodes sont *généralement* traduites comme suit. Tout d'abord, la pré-action est exécutée. Puis, s'il y a une garde, alors on trouve une boucle (while) qui suspend le thread courant (*wait*) tant que la garde n'est pas vraie.

Si la pré-action ou la post-action peuvent potentiellement rendre la garde de certains threads passante alors on insert un appel à *notifyAll(),* de sorte à réveiller tous les threads suspendus qui vont alors tous, chacun à leur tour, re-tester leur garde.

```
.. synchronized meth(..) {
    <pre-action>
     while ! <garde>
           wait() ; **
     <post-action>
    [notifyAll() ;]
}
```

** Par simplicité, on omet le traitement des interruptions au niveau du wait. Lors de la mise en œuvre finale, il faudra remplacer wait() par *try { wait() ; } catch (InterruptedException e) {}*).

Exercice : appliquez ce schéma de traduction pour le problème de l'allocation de ressources. La solution que vous obtenez, appelée Solution Directe, possède les propriétés suivantes :

- **Cohérence** : la solution garantit la cohérence des données. Il n'y a pas plus de ressources allouées que celles disponibles. Par ailleurs, un thread ne sera suspendu lors d'un get() que s'il n'y a plus de ressource libre.
- **Famine** : la solution directe ne garantit pas l'absence de famine. Expliquez pourquoi.
- **Efficacité** : la solution directe n'est pas forcément la solution la plus efficace. Dans le cas présent, identifiez une source d'inefficacité.

3. Construction d'une solution plus efficace

Dans le cas de l'allocation de ressource, on n'a pas besoin de réveiller tous les threads suspendus lorsqu'une ressource se libère, puisqu'au maximum un seul thread pourra obtenir la ressource libérée.

Exercice. Produire une solution où vous ne réveillez qu'un thread lors de la libération d'une ressource.

Questions.
a) Dans cette solution, pourquoi ne peut-on pas remplacer le *while* par un *if* dans la méthode *get()* ? Identifiez un scénario d'exécution qui montre le problème.
b) Cette solution basée sur l'usage de *notify()* à la place de *notifyAll()* garantit-elle l'absence de famine ?

4. Allocation multiple

On considère maintenant le problème de l'allocation *multiple* de ressources, au travers des méthodes *get(n)* (demande de *n* ressource), et *put(n)* (libération de *n* ressource).

Exercice. Tout d'abord modélisez le problème considéré en termes d'un tableau de gardes et d'actions. En déduire ensuite la solution directe.

Questions.

a) Cette solution directe peut-elle produire de la famine ?

b) En remplaçant *notifyAll()* par *notify()* est-ce qu'on élimine la possibilité de famine ?

c) A votre avis, cela a t'il du sens de remplacer *notifyAll()* par *notify()* pour essayer de gagner en efficacité?

5. Rendez-vous

On s'intéresse à nouveau à la mise en œuvre d'une classe permettant de gérer des rendez-vous entre threads. Cette classe fournit une méthode *come()* et un constructeur permettant d'initialiser le nombre de threads attendus au rendez-vous. Programmez la solution directe, en passant par la définition du tableau de Gardes et d'Actions (TGA).

Questions.

a) Cette solution directe fonctionne t'elle ?

b) Peut-elle être optimisée ?

6. Supplément (optionnel)

Variante de l'allocateur de ressources : limitez le nombre de ressources qu'un thread peut s'allouer, notamment au travers d'appels successifs à get(), à une constante donnée (soit MAX).