TD/TP #7 - Moniteurs sans famine (FIFO)

Cours Applications Concurrentes, *Polytech-INFO4*

F. Boyer, N. de Palma, Université Grenoble Alpes, sept 2020

1. Introduction

Dans ce TD/TP vous allez apprendre une méthodologie pour produire une solution FIFO (donc sans famine), à un problème de programmation concurrente. Nous allons appliquer cette méthodologie au problème de l'allocation de ressources. Pour rappel, le problème de l'allocation de ressources consiste à programmer une classe (*Allocator*) permettant à des threads de prendre et de libérer des ressources indifférenciées.

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

2. Le problème

Nous voulons obtenir une solution FIFO, c'est à dire que lorsque des ressources sont libérées, alors les threads doivent obtenir ces ressources dans l'ordre de leur demande. Ainsi, si on a le schéma suivant dans lequel T1 invoque *a.get()* avant T2, qui lui même effectue son invocation avant T3, alors on doit garantir que T1 obtiendra une ressource avant T2, et T2 avant T3.

```
Allocator a = new Allocator(1) ; // 1 ressource
 T1 : a.get() ;
 T2 : a.get() ;
 T3 : a.get()
```

Pour rappel, nous rappelons ci-dessous le code de l'allocateur en version *solution directe*.

```
class Allocator {
  int nfree;
  public void Allocator(int Q) { nfree = Q;}
  public void synchronized get() {
            while (nfree == 0) wait();
             nfree --;
  }
  public void synchronized put() {
            nfree ++;
            notifyAll();
  }
```

Avec cette solution directe, tous les threads bloqués sur *wait()* sont réveillés dès qu'une ressource se libère. On ne peux prédire dans quel ordre ces threads vont reprendre leur exécution et ré-évaluer leur garde.

Même si l'on remplace *notifyAll()* par *notify(),* le thread reveillé par *notify()* peut se faire doubler par un nouvel arrivant à cause du vol de cycle. Donc dans tous les cas, cette solution n'assure pas un ordre FIFO. SI CET ASPECT N'EST PAS CLAIR, REDEMANDEZ DES EXPLICATIONS.

3. Démarche pour obtenir une solution FIFO

La démarche suivie va consister à repartir de la solution directe (qui produit des méthodes qui ont le schéma rappelé ci-dessous), et à appliquer une modification très simple illustrée ci-dessous.

Schéma de solution directe

```
// methode avec garde
.. synchronized meth(..) {
    while ! <garde>
        wait() ; **
    <action>
}
```

```
// methode sans garde
.. synchronized meth(..) {
    <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) {}*).

Schéma de solution FIFO

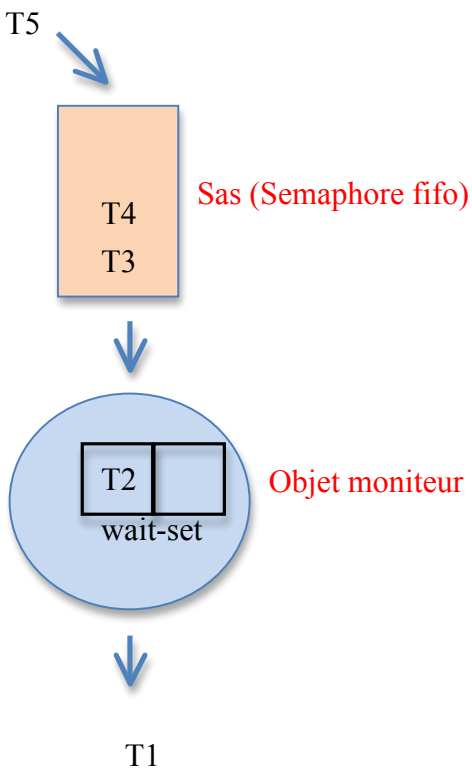
```
1.  Semaphore fifo = new Semaphore(1, true) ;
2.
3.  // methode avec garde
4.  .. meth(..) {
5.   fifo.P();
6.   synchronized(this)
7.       while ! <garde>
8.           wait() ;
9.     <action>
10.  }
11.  fifo.V();
12. …
13. }
```

```
// methode sans garde
.. synchronized meth(..) {
    <action>
    notifyAll() ;
}
…
```

Pour obtenir une solution FIFO, le principe va être de garantir qu'il y a au plus un thread en attente dans le *wait-set* de l'objet courant. Tant que ce thread n'a pas obtenu les ressources qu'il attend, il reste dans le *wait-set* et aucun autre thread ne doit venir se mettre en attente dans ce *wait-set*.

Pour obtenir cela, on va rajouter une sorte de sas de passage devant toute méthode comportant un appel à *wait()*, ce sas libérant les threads au compte-goutte, de manière à garantir qu'il y ait au plus un thread en exécution dans la méthode bloquante. On va utiliser un sémaphore pour implémenter ce sas (ligne 1 dans le listing de gauche donné ci-dessus), ce qui permettra de garantir que les threads sortent du sas dans un ordre FIFO.

Avec cette conception, les threads sont donc maintenant bloqués dans deux files différentes : le *wait-set* du moniteur courant qui contient le prochain thread attendant que sa garde devienne satisfaite (ligne 8), et la file d'attente du sémaphore qui contient les threads suivants (ligne 1).



La figure ci-dessus ce fonctionnement. Le moniteur a laissé passé un thread T1 (sa garde était passante). En revanche, le moniteur a bloqué le thread suivant T2 car sa garde n'est pas passante (ligne 8). Comme T2 est bloqué dans le moniteur, les threads suivants restent bloqués dans le sémaphore fifo (ligne 1).

Lorsque T2 pourra sortir du *wait-set* car sa garde est devenue passante, il libèrera le prochain thread bloqué dans le sémaphore *fifo (ligne 11)*. Ce thread viendra évaluer sa garde (ligne 7), et se bloquer dans le *wait-set* si celle-ci n'est pas passante. **Il restera seul dans le *wait-set***

jusqu'à ce que sa garde devienne passante, ce qui libèrera alors le thread suivant (s'il y en a un) bloqué dans le sémaphore fifo.

Exercice : déroulez pas à pas, sur papier, l'exécution du scénario proposé en suivant le schéma de solution FIFO donné précédemment. Vous devez totalement comprendre ce qui se passe et être capable d'expliquer pourquoi cette solution garantit bien un ordre FIFO.

Question. Dans le schéma de solution FIFO proposé, on a pris soin de ne pas imbriquer l'appel à fifo.P() et fifo.V() dans le bloc synchronisé (on a déplacé la directive *synchronized* pour ce faire). Pourquoi a t'on fait cela à votre avis ? quel problème aurait été posé par la version suivante ?

```
Semaphore fifo = new Semaphore(1, true) ;

// methode avec garde
.. synchronized meth(..) {
 fifo.P();
 while ! <garde>
          wait() ;
 <action>
 fifo.V();
 …
 }
```

Question. Le bloc synchronisé ne contient pas l'instruction fifo.P() (cette instruction ayant été sortie du bloc synchronisé et placée avant le bloc, comme expliqué à la question précédente). La séquence d'instructions globale n'est donc pas atomique. Pourquoi cela ne pose t'il pas de problème ?

4. Application à l'allocateur de ressources unitaire

Exercice : appliquez la démarche expliquée précédemment permettant d'obtenir une solution FIFO à l'allocateur de ressources unitaire.
Une fois que vous avez obtenu une version de la classe Allocator qui garantit un ordre FIFO, déroulez pas à pas, sur papier, l'exécution d'un scénario dans lequel vous avez 3 ou 4 threads qui demandent une ressource à un allocateur qui possède 1 ressource initialement. Vous devez totalement comprendre ce qui se passe et être capable d'expliquer pourquoi cette solution garantit bien un ordre FIFO.

Exercice pratique : si le temps le permet, mettez en œuvre cet allocateur sous Eclipse. Vérifiez que l'ordonnancement FIFO est bien respecté.

5. Application à l'allocateur de ressources multiple

Afin de bien voir à quel point ce schéma de mise en œuvre de solution FIFO est général, appliquez la même démarche pour mettre en œuvre un allocateur de ressources multiple garantissant un ordre FIFO.

Dans un deuxième temps, posez-vous les questions suivantes :

- Dans quelles conditions un tel allocateur a du sens ?
- Quel est le risque principal avec un tel allocateur ?
- Quelles seraient les autres politiques qui pourraient avoir du sens ?

S'il vous reste du temps, vous pouvez essayer de mettre en œuvre un allocateur qui satisfait les plus petites demandes en priorité.