Mini-projet – Moniteurs Java

Cours Applications Concurrentes, *Polytech-INFO4*

F. Boyer, Université Grenoble Alpes, sept 2022

1. Notion de moniteur

De manière générale, un moniteur est une structure qui contient des données et des méthodes exclusives manipulant ces données. Au niveau de sa mise en oeuvre, un moniteur contient un **verrou** (lock) et une **file d'attente** (wait-set), ces structures de données étant généralement ajoutées par le compilateur.

Question 1. A partir de ce rappel, énoncer les principes des moniteurs tels que vus en cours. En particulier, que se passe t'il en début et fin de méthode synchronisée ? que se passe t'il au moment d'un wait() ? que se passe t'il lors de la libération du moniteur ?

2. Moniteurs Java

Comme vu en cours, un moniteur en Java correspond à un objet qui fournit des méthodes synchronisées. Une méthode synchronisée est une méthode associée à la directive *synchronized*. Cette directive peut être définie au grain de la méthode (schéma de gauche ci-dessous) ou à un grain plus fin (bloc de code au sein de la méthode). Dans ce dernier cas, la directive précise la référence de l'objet sur lequel le verrou doit être pris (schéma de droite ci-dessous).

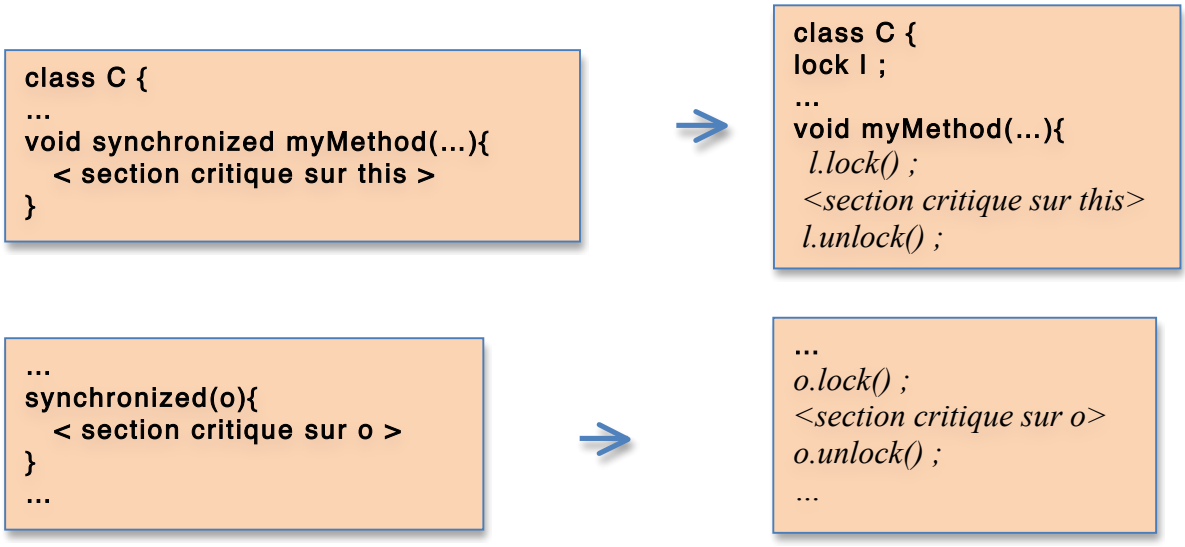
```
public void synchronized myMethod() {
   < section critique sur this >
}
```

```
...
synchronized(o){
   < section critique sur o >
}
...
```

Le compilateur traduit le code précédent en verrouillant l'objet courant en début de bloc synchronisé, et en le déverrouillant en sortie de bloc synchronisé, comme montré ci-dessous.
Pour pouvoir verrouiller un objet, le compilateur ajoute un verrou (*lock)*, ainsi qu'un champ *wait-set* dont nous verrons l'usage plus tard, à la classe définissant l'objet. Le verrou utilisé

est **ré-entrant**, c'est à dire qu'une méthode synchronisée peut invoquer une autre méthode synchronisée sur le même objet.

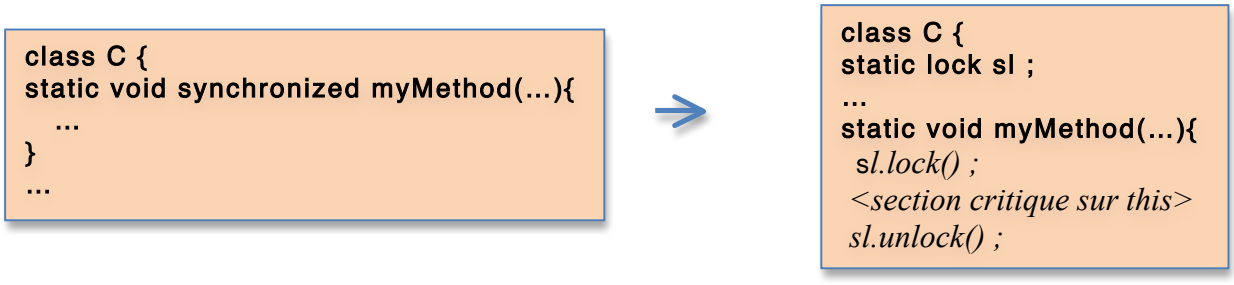
Traduction par le compilateur



3. Moniteurs de type classe vs Moniteurs de type instance

La clause *synchronized* peut aussi être associée à une méthode statique. Dans ce cas, la méthode prendra le verrou sur l'objet classe comme le montre le schéma ci-après.

Traduction par le compilateur



Considérons maintenant la classe C définie comme suit.

```
class C {
  public void synchronized M1(..){ .. }
  public void synchronized M2(..){ .. }
  public static void synchronized M3(..){ .. }
  public static void synchronized M4(..){ .. }
  public void M5(..){ .. }
  public static void M6(..){ .. }
 }
```

Question 3. Soient *a* et *b* deux références vers deux instances distinctes de la classe C. Préciser les invocations qui peuvent (ou pas) être exécutées en parallèle (on suppose que ces invocations sont réalisées par deux threads distincts). On supposera également que les méthodes considérées (M1, .., M6) ne contiennent pas d'appel à *wait()*.

Invocations parallèles ?	O/N
a.M1() // a.M1()	
a.M1() // a.M2()	
a.M1() // b.M1()	
a.M1() // a.M5()	
a.M1() // C.M6()	
a.M1() // C.M3()	
C.M3() // C.M4()	
C.M3() // C.M5()	
a.M1() // a.M1()	

4. Programmation d'un premier moniteur

Exercice pratique. Vous allez maintenant programmer une classe nommée *AllocId* qui délivre des identifiants uniques. Cette classe fournit une méthode statique *int get()* délivrant un nouvel identifiant unique. Cette méthode doit être *thread-safe*, c'est à dire que plusieurs threads sont peuvent invoquer la méthode *get()* en parallèle.

Une méthode thread-safe est une méthode qui peut être invoquée sur un objet O de manière concurrente par plusieurs threads sans que cela engendre un problème de cohérence au niveau de l'objet O. En général, les méthodes d'une classe thread-safe sont implémentées comme des méthodes synchronisées.

Pour tester votre classe, définissez une méthode *main* qui crée un ensemble de threads qui vont enchaîner des appels à la méthode *get(),* de manière à stresser votre système. Vous prendrez soin de vérifier par programme que la valeur finale du compteur correspond bien à celle attendue.
A titre de vérification, exécutez ce programme avec une version non synchronisée de la classe *AllocId*, et constatez que cette fois votre compteur termine avec une valeur erronée.

5. Gestion d'attentes passives (wait-notify)

Tout moniteur Java possède un *wait-set* (une liste de threads bloqués). Le programmeur dispose des trois primitives données ci-après pour manipuler ce *wait-set*. Ces primitives doivent toujours être appelées dans un bloc de code synchronisé (sinon une exception sera levée à l'exécution).

- *wait* : ajoute le thread courant dans le wait-set
- *notify* : réveille un thread bloqué dans le wait-set
- notifyAll : réveille tous les threads bloqués dans le wait-set

Les principes de mise en œuvre de *wait* et *notify* est décrit ci-après. Vous devez impérativement connaître ces principes et les comprendre complètement. Autrement dit, vous devez être capable de reproduire le code de *wait* et *notify* par vous mêmes si on vous le demande.

```
wait :
   put(current-thread, this.wait-set) ;
   current.unlock() ;
   thread-switch();
   this.lock();
```

```
notify:
   Thread t = get(this.wait-set) ;
    if (t != null) put(t, ready-queue) ;
```

Question 5: voici ci-après une classe synchronisée *Account* impliquant des appels à *wait* et *notify*. Décrire en détail ce qui se passe au niveau des structures de type *lock* et *wait-set* dans le scénario suivant :

- On considère l'objet A de classe *Account* dont la balance vaut 10.
- On considère un premier thread T1 qui invoque *A.waitBalanceGreaterThan(20)*.
- On considère un deuxième thread T2 qui invoque quelques secondes plus tard la méthode *A.credit(32)*.

```
class Account {
      int balance=0;
       public void synchronized debit(float f){
            balance -= f ;
       }
       public void synchronized credit(float f){
            balance += f ;
            notifyAll() ;
       }
      public void synchronized waitBalanceGreaterThan(float f) {
            while (balance < f)
               wait() ;
      }
}
```

Exercice pratique. Maintenant on vous demande de tester cela sur machine, en définissant la classe *Account*, et en ajoutant une méthode principale *main* qui crée et démarre le thread T1, s'endort pendant 2 secondes, puis crée et démarre le thread T2.

Comme toute méthode bloquante, *wait()* peut lever l'exception *InterruptedException*. Vous traiterez cette exception de la manière suivante :

```
public void synchronized waitBalanceGreaterThan(float f) {
```

```
        while (balance < f)
            try {wait() ;} catch (InterruptedException e) {}
    }
```

Pour observer ce qui se passe, lancez votre programme en debug, en ayant mis un point d'arrêt au début des méthodes *waitBalanceGreaterThan()* et *credit()*. Mettez également un point d'arrêt dès les premières instructions de votre méthode *main()*.

Au démarrage, vous ne voyez dans le debugger qu'une seule pile d'appel (*stack trace* en anglais), correspondant à celle de votre thread principal. Avancez en pas à pas sur le programme suivi par ce thread principal.

Après avoir créé et démarré le premier thread (T1) qui invoque la méthode *waitBalanceGreaterThan*(20), vous pourrez observer la pile du thread T1. Ce thread va se suspendre sur le point d'arrêt associé à la méthode *waitBalanceGreaterThan*. Vous pouvez voir au niveau de la stack trace affichée par Eclipse une petite clé, indiquant que le thread T1 est dans le moniteur de classe Account.

Sélectionnez le thread T1 au niveau de la stack trace affichée dans Eclipse (cliquez sur la ligne *Account.waitBalanceGreaterThan(float))* et avancez en pas à pas. Vous allez voir que votre thread ne pourra pas avancer plus loin que l'appel à *wait*(), cet appel étant bloquant.

Reprenez le pas à pas sur votre thread principal, jusqu'à ce que le thread T2 soit démarré. Ce thread T2 va se suspendre sur le point d'arrêt associé à la méthode *credit*(). Sélectionner ce thread T2 et faites le avancer en pas à pas jusqu'à ce qu'il exécute *notifyAll*(). Observez qu'après l'exécution de cette instruction, le thread T1 peut à nouveau avancer dans son exécution.

Par défaut, les points d'arrêt que vous définissez n'engendrent que la suspension du thread qui exécute l'instruction associée au point d'arrêt. Si vous voulez définir des points d'arrêt qui engendrent l'arrêt de l'ensemble des threads, vous pouvez le faire en modifiant les propriétés du point d'arrêt, mais cela reste une fonctionnalité moins fréquemment utilisée.

Globalement, le debugger permet donc de visualiser et contrôler les exécutions des différents threads, mais sachez toutefois que l'exécution de threads concurrents reste beaucoup plus complexe à suivre qu'un programme séquentiel.

6. Gestion des interruptions

Tout thread bloqué sur un appel bloquant (wait(), sleep(), etc) peut être interrompu, par un autre thread applicatif via la méthode *interrupt()* définie par la classe Thread, ou par la machine virtuelle (qui se réserve le droit d'interrompre un thread pour des raisons internes).

Si un thread est interrompu alors qu'il est en attente dans une méthode bloquante, il sort de la méthode bloquante par la réception de l'exception *InterruptedException*. Sinon le flag *interrupted* du thread est positionné à vrai. Tout thread peut tester ce flag à tout moment via la méthode *isInterrupted*().

En général, la programmation d'une séquence de type [if <condition > wait() ;] prendra la forme suivante pour résister aux interruptions:

```
while (<condition>) {
      try {
          wait() ;
       } catch (InterruptedException e) {}
 }
```

7. Classe RDV

Exercice pratique. Programmer une classe nommée *Rdv*, dont le constructeur permet de préciser le nombre de threads attendus au rdv. La méthode *come()* permet à un thread d'arriver au rdv. Tout thread arrivant au rendez-vous est mis en attente tant que les autres threads attendus ne sont pas tous arrivés.

Exemple d'usage de la classe Rdv :

```
…
Rdv rdv = new Rdv(3) ;
MyThread mt1 = new MyThread(rdv).start() ;
MyThread mt2 = new MyThread(rdv).start() ;
MyThread mt3 = new MyThread(rdv).start() ;
…

class MyThread extends Thread {
    Rdv rdv ;
    public MyThread(Rdv rdv) { this.rdv = rdv ;}
    public void run() { … ; rdv.come() ; …}
}
```

9. Propriétés importantes des moniteurs Java qu'il faut absolument retenir

- **Ré-entrance**

Les moniteurs Java sont ré-entrants, c'est à dire qu'un thread ayant réussi à entrer dans un bloc synchronisé d'un moniteur peut entrer dans d'autres blocs synchronisés du même moniteur. En d'autres termes, une méthode synchronisée peut faire appel à d'autres méthodes synchronisées du même moniteur. Le seul point auquel il faut faire attention est le cas dans lequel une méthode synchronisée invoque une autre méthode synchronisée qui contient des appels à *wait()* (voir ci-après).

- **Imbrications de clauses synchronisées avec appel à wait()**

Lorsqu'un thread se bloque, il ne libère que le verrou de l'objet courant (voir schéma de traduction du *wait()* donné en section 6 de cette note). En cas d'imbrications de clauses synchronized, les verrous de plus haut niveau restent donc pris. De ce fait, attention aux interblocages quand on imbrique des blocs synchronisés.

→ Par principe, on évitera d'imbriquer des blocs synchronisés s'il y a des appels à wait.

- **Vol de cycle**

Tout thread réveillé par un notify() doit ré-acquérir le verrou pour poursuivre son exécution dans le moniteur courant. Dans cette étape de ré-acquisition du verrou, il peut être en concurrence (et donc se faire doubler par) des nouveaux arrivants. C'est ce que l'on appelle le vol de cycle.