INSA
CENTRE VAL DE LOIRE

POO Avancée : Java

Exclusion mutuelle de threads

Salwa SOUAF

Outline

Les problèmes liés à la concurrence

L’exécution concurrente de threads ayant accès au même espace mémoire peut provoquer des écriture/lecture entrelacées rendant les données incohérentes ou faussant le cours "voulu" de l’exécution. On tente alors de protéger les données des accès concurrents en excluant les threads mutuellement. Techniquement, il s’agit de poser des locks ou des moniteurs.
L’exemple classique consiste à considérer deux threads et un objet stockant des valeurs dans un tableau (par exemple à l’aide de la classe Vector). Un des threads réalise des écritures dans cet objet pendant que l’autre réalise des parcourts. Suivant la manière dont le scheduler donne la main aux threads et suivant la fréquence à laquelle chaque thread accède à l’objet, il est possible que le thread réaliser une opération d’écriture pendant que l’autre thread est en train de réaliser un parcourt. Généralement, cela génère la levée de l’exception ConcurrentModificationException.

Outline

Locks et moniteurs

Les systèmes préemptifs demandent l'utilisation de lock afin d'éviter les problèmes posés par un accès concurrent à une même ressource partagée. On peut distinguer deux méthodes équivalentes, mais ayant des sémantiques différentes, permettant de sérialiser les accès à une ressource. La première déjà évoquée est le lock qui permet (en l'associant à une donnée) de s'assurer qu'il n'y aura pas d'accès concurrent à cette donnée partagée. La procédure est assez simple :

- Acquisition du lock
- Utilisation des données partagées
- Relâchement du lock Dans cette approche on restreint l'accès à la donnée, il faut donc prendre le lock avant chaque utilisation de cette donnée. Dans Java 1.5, on dispose de la classe Lock pour cela.

La deuxième approche (moniteur) ne consiste plus à restreindre l'accès à une donnée mais au code qui modifie cette donnée. C'est la solution retenue par pour réaliser la protection des données partagées. Dans cette approche, on sérialise l'accès à une portion de code.
Un moniteur se pose à l'aide du mot clef synchronized: on dit aussi que le bloc est synchronisé. Si c'est la méthode tout entière sur laquelle on pose un moniteur sur this, la méthode est dite "synchronisée".
Enfin, un objet peut être totalement synchronizé: toutes ses méthodes le sont.

Locks et moniteurs

Exemple de blocs synchronisés

L'exemple ci-dessous montre comment réaliser la gestion d'un tableau dynamique à accès concurrents à l'aide de blocs (ou méthodes) synchronisés.

```
public class TableauDynamique {
  private Object[] tableau; // le conteneur
  private int nb; // la place utilisée

  public TableauDynamique (int taille) {
    tableau = new Object[taille];
    nb = 0; }
  public synchronized int size() {
    return nb; }
  public synchronized Object elementAt(int i) throws NoSuchElementException {
    if (i < 0 || i = nb)
      throw new NoSuchElementException();
```

```
    else
      return tableau[i];
  }
  public void append(Object x) {
  Object[] tmp = tableau;
  synchronized(this) { // allouer un tableau plus grand si nécessaire
    if (nb == tableau.length) {
      tableau = new Object[3*(nb + 1)/2];
    for (int i = 0; i < nb; ++i)
      tableau[i] = tmp[i];
    }
    tableau[nb] = x;
    nb++;
  }
}}
```

Méthodes synchronisées

Une méthode synchronisée peut appeler une autre méthode synchronisée sur le même objet sans être suspendue. En effet, le thread courant possède le moniteur sur l'instance de la classe dont on exécute la méthode: il peut donc appeller une autre méhode du même objet, ce moniteur étant déjà acquis.
Les méthodes statiques peuvent aussi être synchronisées mais la synchronisation se fait sur l'objet de type Class. Si une méthode non statique veut synchroniser son accès avec une méthode statique elle doit se synchroniser relativement au même objet. Elle doit donc contenir une construction de la forme : synchronized(this.getClass())....

Locks et moniteurs

Méthodes synchronisées

Le choix de l'objet sur lequel le moniteur est posé est primordial. Choisir this comme objet pour l'appel à synchronized(this) sans réfléchir et vérifier que l'exclusion est bien effective est une grossière erreur. En général, l'objet à passer à l'appel à synchroniser est l'objet à protéger et qui sera commun à tous les threads. Il est aussi possible de créer un objet ad-hoc spécialement concu pour la synchronisation. Dans un certain sens, on revient alors à l'utilisation des classes implémentants Lock. Typiquement, un objet de type Ressource peut être utilisé par un thread pour exclure un autre threads:

```
// Quelque part dans le thread principal
Ressource r = new Ressource();
// Dans chaque portion de code exécutée dans des threads:
synchronized(r) {
  // ici, j'exclu les threads souhaitant poser un moniteur sur r
  ... }
```

Locks et moniteurs

Locks

L'interface Lock fournit les primitives nécessaire pour manipuler différentes classes de locks. Dans Java 1.5, ces classes ont un comportement spécifique qu'il faut adapter au type de modèle traité. Nous donnons par la suite deux exemples de locks particuliers.

```
Lock l = ...;
l.lock();
try {
    // access the resource protected by this lock
} finally {
    l.unlock();
}
```

Locks et moniteurs

Locks

ReentrantLock

Ce lock est dit réentrant dans le sens ou un thread possédant déjà ce lock et le demandant à nouveau ne se bloque pas lui-même. Ce type de blocage ne peut pas survenir avec un moniteur puisque un moniteur déjà posé sur une ressource est considéré comme acquis si un autre appel survient essaynt un moniteur sur cette même ressource.

ReadWriteLock

Ce lock fournit une implémentation permettant d'avoir une politique d'accès à une ressources avec plusieurs lecteurs et un unique écrivain. Les accès en lecture sont alors très efficace car réellement concurrents.

Outline

Liveness / Vivacité

Problème de vivacité ou liveness

A cause de l'exclusion mutuelle qui tente d'assurer que rien de faux n'arrive, il est courant que rien n'arrive du tout: c'est le problème de vivacité ou liveness.
Les problèmes de liveness peuvent être de quatre types :

- Famine (contention) Bien que la thread soit dans une état ou elle puisse s'exécuter, une autre thread (plus prioritaire par exemple) l'empêche toujours de s'exécuter.
- Endormissement (dormancy) Un thread est suspendu mais n'est jamais réveillée.
- Interblocage (deadlock) Plusieurs threads s'attendent de façon circulaire avant de continuer leur exécution.
- Terminaison prématurée Un thread (à cause d'un ordonnancement particulier) reçoit un stop() prématurément.

Le problème de liveness le plus courant est le problème de deadlock. Un thread1 attend une "chose" que doit libérer un thread2 qui attend lui-même une chose que doit libérer le thread1. Une stratégie simple (malheureusement pas toujours applicable) pour éviter les deadlocks consiste à numéroter les "choses" à attendre et à toujours les prendre dans le même ordre.

Liveness / Vivacité

Exemple de deadlock

Le système de lock peut provoquer des interblocages ou deadlocks. L'exemple suivant utilise de objets de type Cell, cell1 et cell2. Si le thread 1 exécute swapValue sur cell1, il prends le moniteur sur cell1. De même le thread 2 peut prendre le moniteur sur cell2 où cell2=other dans le code cell1. Dans ce cas, à l'instruction other.getValue(), cell1 attend d'avoir le moniteur sur cell2 et inversement...

```
class Cell {
  private long value_;

  synchronized long getValue() { return value_;}
  synchronized void setValue(long v) {value_ = v;}

  synchronized void swapValue(Cell other) {
    long t = getValue();
    long v = other.getValue();
    setValue(v);
    other.setValue(t);
   }
  }
```

Liveness / Vivacité

Solution dite d'ordre

Une solution classique dite "d'ordre" permet de résoudre ce problème. Elle utilise la notion de précédence des locks: il s'agit de toujours prendre le lock sur l'objet ayant le plus grand hashcode, avant de passer au suivant.

```
void swapValue(Cell other) {
 if (other == this) return; // alias check

 Cell fst = this; // order via hash codes
 Cell snd = other;
 if (fst.hashCode() > snd.hashCode()) {
   fst = other;
   snd = this;
 }
 synchronized(fst) {
   synchronized (snd) {
     long t = fst.value;
     fst.value = snd.value;
     snd.value = t;
   }
 }
}
```

Liveness / Vivacité

Résoudre la famine ou l'endormissement

Une autre approche qui peut souvent porter ces fruits pour éliminer les problèmes de famine ou d'endormissement est la stratégie "diviser pour régner". Plutôt que d'avoir un unique moniteur, celui-ci est subdivisé en plusieurs sous-moniteurs contrôlant chacun l'accès à une ressource particulière. Cela permet de minimiser le temps d'attente des threads et augmente l'efficacité d'exécution du programme.
Malheureusement, la multiplication des moniteurs favorise l'apparition de deadlocks. Si un thread doit poser plusieurs moniteurs sur plusieurs ressources, la possibilité d'un deadlock apparait. Une stratégie de résolution consiste à ne pas attendre un moniteur s'il est déjà pris: si un moniteur est occupé, on relâche alors tous les moniteurs déjà pris et on diffère l'action à réaliser.
Enfin, la meilleure stratégie pour éviter tout problème de liveness est de supprimer au maximum les exclusions mutuelles, quand cela est possible. Avant de supprimer l'exclusion mutuelle d'une partie de code il est très important de penser aux deux règles suivantes :

- Ne jamais avoir d'idée préconçue sur la progression relative de deux threads. En particulier toujours vérifier si le code est valide dans le cas ou l'autre thread n'a même pas démarré ou si il est déjà terminée.
- Supposer que le scheduler peut libérer le processeur du thread courant à n'importe quel point du code.

Liveness / Vivacité

Les sémaphores

Les sémaphores permettent d'étendre la notion de lock en introduisant la manipulation de jetons (ou de permissions). Un sémaphore permet d'acquérir et de relâcher un jeton. Si un thread ne peut acquérir de jetons, il reste bloqué sur l'acquisition jusqu'à ce qu'un autre thread relâche un jeton. D'autre primitives permettent notamment de connaître le nombre de jeton disponibles, d'essayer d'acquérir un jeton si possible mais de manière non bloquante.
Il n'y a aucune contrainte sur l'identité des objets et des threads qui acquièrent ou relâchent un jeton d'un sémaphore. C'est donc un système très flexible mais générateur de bugs, puisqu'on peut très facilement oublier de relâcher un jeton et bloquer d'autres threads. Pour éviter ce genres de problèmes, on peut déjà essayer d'acquérir et de relâcher des jetons au sein d'une même classe pour éviter la dispersion du code de gestion du sémaphore dans de multiples objets.

Les sémaphores

Notons enfin qu'un sémaphore ayant un seul jeton (ou permis) est appelé un sémaphore binaire et peut servir de lock d'exclusion mutuelle. La différence entre un sémaphore binaire et un lock réside dans le fait qu'un sémaphore binaire peut être relâché par un thread différent, ce que ne permet pas toujours de faire un lock (Lock n'est qu'une interface). Ceci peut-être utile dans certains cas pour éviter des deadlock.

```
Semaphore s = new Semaphore(3, true);
s.acquire() // prend un jeton
s.release() // relache un jeton
```

Liveness / Vivacité

Exemple d'utilisation de sémaphores

Dans l'exemple suivant 100 jetons sont disponibles. La classe Pool fournit des objets à l'utilisateur mais n'est pas censé en fournir "trop". Il serait par exemple interdit de donner le 101ème objet contenu dans le pool d'objet. En programmation non-concurrente, un simple test suffirait. En programmation concurrente, l'utilisation d'un sémaphore devient obligatoire !

```
class Pool {
   private static final MAX_AVAILABLE = 100;
   private final Semaphore available =
                  new Semaphore(MAX_AVAILABLE, true);

   public Object getItem() throws InterruptedException {
     available.acquire();
     return getNextAvailableItem();
   }

   public void putItem(Object x) {
     if (markAsUnused(x))
       available.release();
   }
}
```

Outline

Synchronisation

Pour les applications concurrentes il est souvent très important de pouvoir synchroniser des threads entre eux. Ceci particulièrement utile pour s'assurer qu'un autre thread est bien dans un certain état (terminé en particulier).
Java propose un mécanisme d'attente/notification. Les primitives suivantes sont des méthodes de la classe java.lang.Object qui sont utilisées pour la synchronisation. Elles doivent être appelées sur un objet associé à un moniteur détenu au moment de l'appel par le thread courant:

- public void wait() throws InterruptedException suspend l'activité de la thread courante, "libère" le moniteur et attend une notification sur le même objet. Quand la thread a reçu une notification elle peut continuer son exécution dès qu'elle a "repris" le moniteur du bloc synchronisé courant.
- public final native void wait(long timeout) throws InterruptedException est équivalente à la précédente mais l'attente est bornée par un paramètre de timeout en millisecondes. Avant de pouvoir reprendre son exécution après un timeout il est toujours nécessaire de reprendre le moniteur.
- public void notify() envoie une notification à une thread en attente wait() sur le même objet.
- public void notifyAll() envoie une notification à toutes les threads en attente wait() sur le même objet.

Exemple de synchronization

L'exemple suivant montre comment temporiser l'incrémentation, lorsque celle-ci n'est pas possible. Si l'utilisateur appelle inc() et que le compteur est déjà au maximum MAX, l'objet appelle tout d'abord attendIncrementable() qui ne rend la main que si l'on peut incrémenter. Si cela n'est pas possible, la méthode appelle wait() pour mettre "en pause" le thread courant, libérer l'objet et attendre d'être notifié d'un changement: en l'occurence, on attend une décrémentation.

Synchronisation

Exemple de synchronization

```
public interface Compteur {
public static final long MIN = 0; // minimum
public static final long MAX = 5; // maximum

public long value(); // valeur entre MIN et MAX
public void inc(); // incremente si value() < MAX
public void dec(); // decremente si value() MIN }

public class CompteurConcurrent implements Compteur {
 protected long count = MIN;
 public synchronized long value() {
 return count; }

 public synchronized void inc() {
 attendIncrementable();
 setCount(count + 1); }
 public synchronized void dec() {
 attendDecrementable();
 setCount(count - 1); }
 protected synchronized void setCount(long newValue) {
 count = newValue;
 notifyAll(); }
 protected synchronized void attendIncrementable() {
 while (count >= MAX)
 try { wait(); } catch(InterruptedException ex) {}; }
 protected synchronized void attendDecrementable() {
 while (count <= MIN)
 try { wait(); } catch(InterruptedException ex) {};
 }
}
```

Timers

Les Timers permettent de planifier des taches régulières. Il s'agit de réveiller des objets implémentant TimerTask à intervalles réguliers. On utilise la classe Timer pour spécifier les intervalles de valeurs. Toutes les tãches planifiées s'exécutent alors dans le même processus léger. On peut annuler un timer avec la méthode cancel() de Timer ou annuler une tãche planifiée en utilisant cette fois celle de TimerTask.

Synchronisation

Timers

```
import java.util.*;
public class DateTask extends TimerTask {
  String msg;
  public DateTask(String msg) { this.msg = msg; }
  public void run() {
    System.out.println(Thread.currentThread().getName() +
                       " " + msg + ": " + new Date());
  }
}

public class TimerExample {
  public static void main(String[] args) {
    Timer timer = new Timer();
    DateTask task0 = new DateTask("task0");
    DateTask task1 = new DateTask("task1");
    DateTask task2 = new DateTask("task2");
    Calendar cal = Calendar.getInstance();
    cal.set(Calendar.HOUR_OF_DAY,17);
    cal.set(Calendar.MINUTE,14);
    cal.set(Calendar.SECOND,0);
    Date givenDate = cal.getTime();
    //task0: dès maintenant, toutes les 5 secondes
    timer.schedule(task0, 0, 5000);
    //task1: départ dans 2 secondes, toutes les 3 secondes
    timer.schedule(task1, 2000, 3000);
    //task2: une seule fois à la date fixée
    timer.schedule(task2, givenDate);
    System.out.println(Thread.currentThread().getName()
                                          + " terminé!");
  }
```

Outline

Thread safe collections

Les accès concurrents à des collections provoquent souvent la levée de l'exceptions ConcurrentModificationException. L'idée principale des collections thread-safe est d'implémenter des classes robustes face à des accès concurrents mais faiblement consistantes au niveau de la cohérence des données. Une facon simple d'implémenter une collection thread-safe est de garantir la synchronisation des opérations, à la main, ou à l'aide de wrappers:

```
public static <T> Collection<T> synchronizedCollection(Collection<T> c)
```

Il ne faut cependant pas oublier de protéger les itérateurs pendant toute la durée de l'itération:

```
Collection c = Collections.synchronizedCollection(myCollection);
   ...
synchronized(c) {
    Iterator i = c.iterator(); // Must be in the synchronized block
    while (i.hasNext())
       foo(i.next());
}
```

Thread safe collections

Exemple de création d'un wrapper thread-safe

Pour illustrer comment un code peut être rendu thread-safe, on considère la servlet suivante:

```
public class UnsafeGuestbookServlet extends HttpServlet {
    private Set visitorSet = new HashSet();
    protected void doGet(HttpServletRequest httpServletRequest,
                         HttpServletResponse httpServletResponse)
                         throws ServletException, IOException {
        String visitorName = httpServletRequest.getParameter("NAME");
        if (visitorName != null)
            visitorSet.add(visitorName);
    }
}
```

modifiée en remplacant l'attribut privé par:

```
private Set visitorSet = Collections.synchronizedSet(new HashSet());
```

Ces wrappers introduisent cependant des difficultés au niveau des performances d'utilisation des collections. Si l'accès est fréquent ou si l'espace mémoire utilisé est conséquent, l'utilisation de copies temporaires fait augmenter le nombre de traitements et l'espace mémoire occupé.

Thread safe collections

Wrappers thread-safe et faiblement consistants

L'explication précédente se base sur l'idée que le plus simple est d'imposer l'exclusion mutuelle sur l'accès à une collection; il suffit pour cela d'ajouter un synchronized sur les méthodes de classes pour garantir un comportement thread-safe. Ceci étant, il faut considérer les deux problèmes suivants:

- De nombreuses opérations sont composées (exemple: création d'un itérateur, puis appel à next() pour le parcourt) et requierent donc des mécanismes de synchronization plus complexes.
- Certains opérations, comme get() peuvent supporter la concurrence, si l'on autorise la lecture multiple. Pour ces raisons, Java 1.5 introduit un certains nombres de wrappers qui sont thread safe mais faiblement consistants.

Le premier exemple que l'on peut prendre, commun à toutes les collections, est l'itérateur faiblement consistant (weakly consitent). Lors de l'appel à next(), si un nouvel élément à été ajouté entre temps, il sera ou ne sera pas retourné par next(). De même, si un élément est enlevé entre deux appels à next(), il ne sera pas retourné pour les prochains appels (mais il a pu avoir été déjà retourné). Ceci étant, dans tous les cas, l'appel à next() ne levera pas l'exceptions ConcurrentModificationException.

Thread safe collections

Listes, Vecteurs, HashMap, Queues

Les mêmes idées d'accès robustes sont implémentées dans les structures de données classiques, au niveau des primitives d'accès. La classe CopyOnWriteArrayList permet de créer un wrapper d'accès à un liste (ArrayList) ou à un vecteur (Vector). Ce système de wrapper évite de proposer de nouvelles classes, remplaçant ArrayList et Vector et met en place le comportement suivant: lorsqu'un accès d'écriture modifie la structure, une copie de la liste ou du vecteur est créée et les itérations de parcours en cours se font sur cette copie. Lorsque toutes les itérations en cours sont terminées, la copie est alors détruite.
De même, un système de wrapper permet de transformer une HashMap en ConcurrentHashMap. Les opérations multiples sont autorisées, en lecture et en écriture et les itérateurs retournés sont weakly consistent. Ce wrapper est diamétralement opposé à Collections.synchronizedMap qui est une implémentation ou chaque méthode est synchronizée garantissant l'exclusion mutuelle. Pour les ensembles, un système de wrapper permet de transformer un HashSet en set à accès concurrent.
Pour les queues, la classe ConcurrentLinkedQueue implémente une version thread safe d'une queue FIFO. Java 1.5 introduit des queues bloquantes, ce qui permet par exemple de faire attendre un producteur, lorsqu'un consommateur est trop lent par rapport au producteur. Ces classes implémentent l'interface BlockingQueue.