Programmation Concurrente

Karima Ennaoui

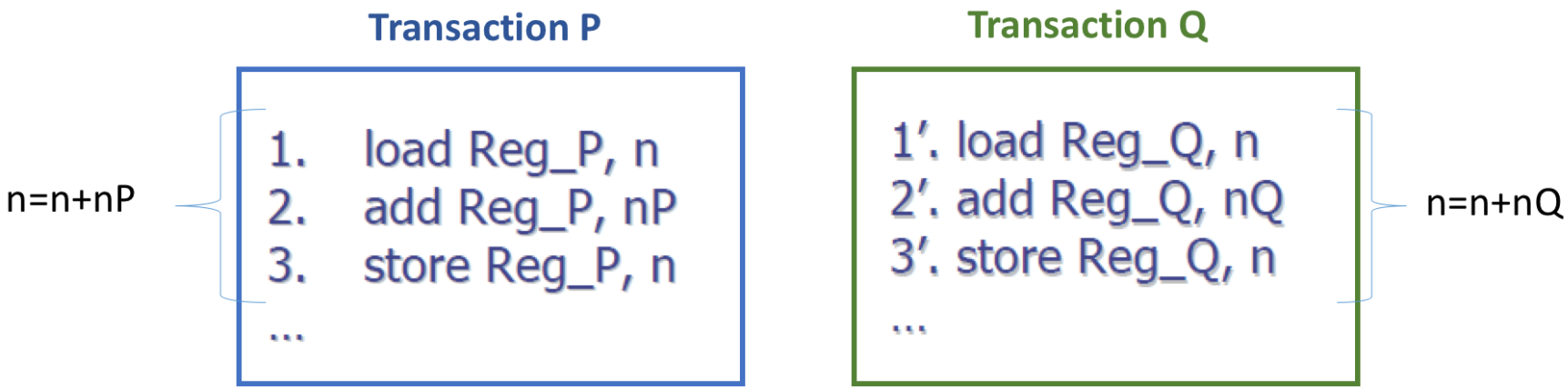
karima.ennaoui@insa-cvl.fr

INSA CVL, 4A STI

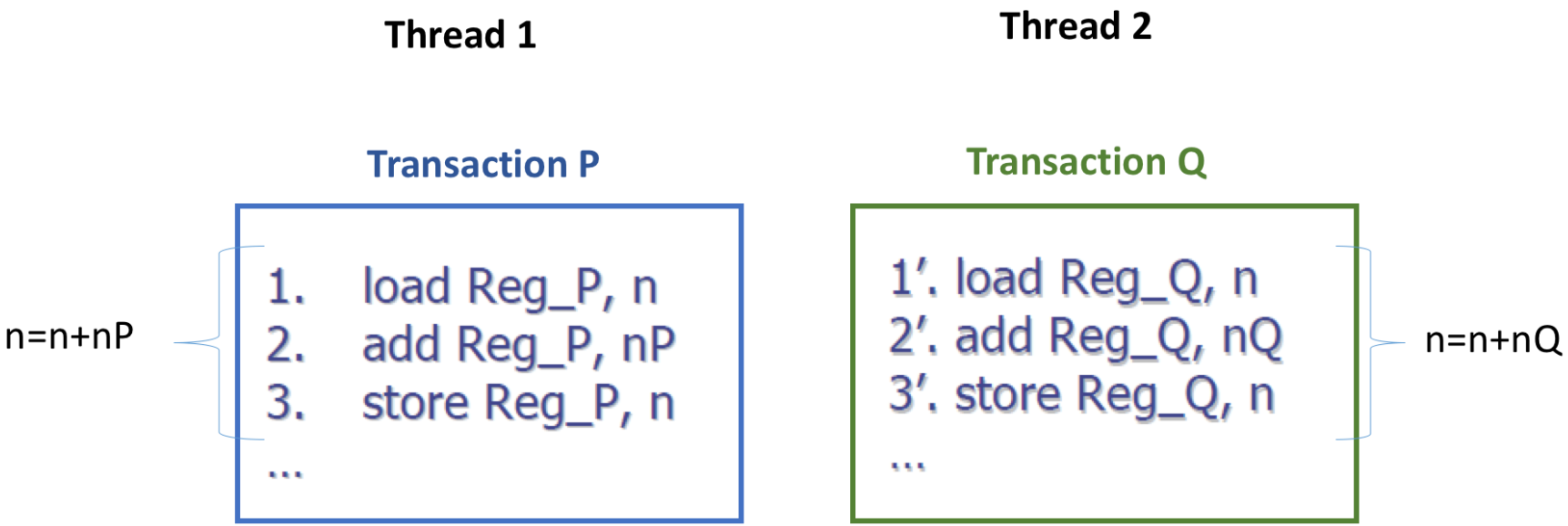
Bibliographie: Java concurrency in practice, Brian Goetz, with Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea, Addison Wesley 2006
https://docs.oracle.com/javase/tutorial/essential/concurrency/
Inspiré du cours de Jean-François Lalande et de Françoise Baude-Dreysse

Sunday 6th January, 2019

Commençant par un exemple...



Exemple:


Thread 1
Thread 2
Transaction P
Transaction Q
n=n+nP
1. load Reg_P, n
2. add Reg_P, nP
3. store Reg_P, n
...
1'. load Reg_Q, n
2'. add Reg_Q, nQ
3'. store Reg_Q, n
...
n=n+nQ

Exemple:

1, 2, 3, 1′, 2′, 3′
1′, 2′, 3′, 1, 2, 3

n=n+nP+nQ
OK !

Transaction P

1. load Reg_P, n
2. add Reg_P, nP
3. store Reg_P, n

...

Transaction Q

1′. load Reg_Q, n
2′. add Reg_Q, nQ
3′. store Reg_Q, n

...

Exemple:

1, 2, 3, 1′, 2′, 3′
1′, 2′, 3′, 1, 2, 3

n=n+nP+nQ
OK !

1, 1′, 2, 2′, 3, 3′
1, 2, 1′, 2′, 3, 3′
1, 1′, 2′, 2, 3, 3′
1, 2, 1′, 3, 2′, 3′
1, 1′, 2, 3, 2′, 3′

n=n+nQ
PBM : nP perdu !

Transaction P

1. load Reg_P, n
2. add Reg_P, nP
3. store Reg_P, n

...

Transaction Q

1′. load Reg_Q, n
2′. add Reg_Q, nQ
3′. store Reg_Q, n

...

Exemple:

1, 2, 3, 1′, 2′, 3′
1′, 2′, 3′, 1, 2, 3

n=n+nP+nQ
OK !

1, 1′, 2, 2′, 3, 3′
1, 2, 1′, 2′, 3, 3′
1, 1′, 2′, 2, 3, 3′
1, 2, 1′, 3, 2′, 3′
1, 1′, 2, 3, 2′, 3′

n=n+nQ
PBM : nP perdu !

1′, 1, 2′, 2, 3′, 3
1′, 2′, 1, 2, 3′, 3
1′, 1, 2, 2′, 3′, 3
1′, 2′, 1, 3′, 2, 3
1′, 1, 2′, 3′, 2, 3

n=n+nP
PBM : nQ perdu !

Transaction P

1. load Reg_P, n
2. add Reg_P, nP
3. store Reg_P, n

...

Transaction Q

1′. load Reg_Q, n
2′. add Reg_Q, nQ
3′. store Reg_Q, n

...

Exemple:

- Il faut que 3 précède 1' ou 3' précède 1, autrement dit, il faut que le thread 1 attend que le thread 2 finisse les trois opérations de la transaction P ou vice versa.
- Ceci est du au fait que les deux threads partagent la même ressource, qui est la case mémoire de n.
- On dit que thread 1 et 2 sont en concurrence pour l'utilisation de la ressource et il faudrait les synchroniser afin que le code soit **correct**.

Concurrence:
La concurrence inter-threads designent ce contexte la rivalité entre eux, afin d'accèder à une ressource partagé en même temps. Il peut s'agir de:

- Ressources physiques (du hardware: processeur, caméra...), ou
- Ressources logiques (Zone mémoire, fichier, une base de données...)

Concurrence:
Cette concurrence, si pas bien gérée, pourrait causer des problèmes au niveau de:

- La correction du code (entrelacement des read/write rendant les données dans un état incohérent), ou
- Sa performance (blocage inter-thread afin d'accéder à une ressource).

Section Critique

Tout d'abord, il faut:

- Repérer les blocs d'opérations, dites **Sections Critiques**, où l'exécution doit être faite d'une manière non entrelacée.
- Ces sections doivent être capable de s'exécuter d'une manière "atomique", sous l'hypothèse qu'elles ont accès exclusive à la ressource manipulée dedans.
- C'est ce qu'on appelle l'exécution en exclusion mutuelle.

Puis, il faudrait Proteger cette section en ajoutant un protocole d'entrée et de sortie.

Propriétés d'une Solution:

- Exclusion mutuelle: Il y a **au plus** une entité en section critique.

Propriétés d'une Solution:

- Exclusion mutuelle: Il y a **au plus** une entité en section critique.
- Pas de Famine: Garantir à tout thread qu'il aura accès à la SC qu'il demande.

Propriétés d'une Solution:

- Exclusion mutuelle: Il y a **au plus** une entité en section critique.
- Pas de Famine: Garantir à tout thread qu'il aura accès à la SC qu'il demande.
- Pas d'interblocage (fatale): si plusieurs threads demandent accès en même à une SC non alloué, un doit toujours être capable d'y accéder.

Propriétés d'une Solution:

- Exclusion mutuelle: Il y a **au plus** une entité en section critique.
- Pas de Famine: Garantir à tout thread qu'il aura accès à la SC qu'il demande.
- Pas d'interblocage (fatale): si plusieurs threads demandent accès en même à une SC non alloué, un doit toujours être capable d'y accéder.
- Condition de progression: Un thread en dehors de la SC ne doit pas ralentir un autre à y accéder.

Propriétés d'une Solution:

- Exclusion mutuelle: Il y a **au plus** une entité en section critique.
- Pas de Famine: Garantir à tout thread qu'il aura accès à la SC qu'il demande.
- Pas d'interblocage (fatale): si plusieurs threads demandent accès en même à une SC non alloué, un doit toujours être capable d'y accéder.
- Condition de progression: Un thread en dehors de la SC ne doit pas ralentir un autre à y accéder.
- Code identique: souhaitable pour la simplicité.

Types de Solutions:
On distingue deux solutions équivalentes mais avec deux sémantiques différentes:

- Solutions avec arbitrage: accès géré par les différentes entités en attente active.
- Solution sans arbitrage: accès géré par un appel implicite à un arbitre, ce qui permet une attente passive des threads.

Solutions sans Arbitrage: Principe

Le principe est que chaque thread attend la satisfaction d'une action pour entrer dans la section critique. Afin de pouvoir tester en permanence la condition, cette attente doit être active (pas de blocage).

Tant que (condition indique SC non libre) **Faire**
rien ou sleep(court délai)
Fintantque
<section de code critique>
Modifier condition pour refléter SC libre
<section de code non critique>

Solutions sans Arbitrage: Algorithme Naîf

T1

```
Répéter
    Tant que (Tour = 1) Faire
        sleep(délai);
    Fintantque
    <section critique>
    Tour :=2
Jusqu'à faux
```

T2

```
Répéter
    Tant que (Tour = 2) Faire
        sleep(délai);
    Fintantque
    <section critique>
    Tour :=1
Jusqu'à faux
```

Exclusion Mutuelle	✓
Absence de blocage	✓
Condition de progression	✗
Absence de famine	✓
Code identique	✗

Solutions sans arbitrage: Algorithme de Peterson

T1

```
...
    Drapeau1 = vrai ; Tour:=2
    Tant que (Drapeau2  ET Tour=2)
    Faire   sleep(délai);
    Fintantque
    <section critique>
    Drapeau1 := faux
...
```

T2

```
...
    Drapeau2 = vrai ; Tour:=1
    Tant que (Drapeau1  ET Tour=1)
    Faire  sleep(délai);
    Fintantque
    <section critique>
    Drapeau2 := faux
...
```

Exclusion Mutuelle	✓
Absence de blocage	✓
Condition de progression	✓
Absence de famine	✓
Code identique	✗

Tour : variable commune, initialement =1 (par ex)
Drapeau1, Drapeau2, initialement = faux
(quand tour =i, Pi peut entrer en SC)

Solutions sans arbitrage: Analyse

- Solutions coûteuse au niveau de temps de CPU
- Solutions Techniques, sources d'erreurs si on oublie de libérer la SC.
- Ordonnanceur n'est pas explicitement sollicité, ce qui pourrait engendrer des problèmes d'équité, où un thread garde le CPU pour longtemps.

Solutions avec arbitrage: Principe

Répéter
...
① Est-ce que je peux entrer en SC (je demande ceci à l'arbitre) ?
② Si NON alors je me bloque ...
<section critique>
Je relâche la SC (en le disant à l'arbitre), et la question se pose :
Est-ce qu'un processus est bloqué ?
Si OUI alors le réveiller (ou Ordonner un réveil)
Jusqu'à faux

- Si la demande d'un thread pour entrer dans une SC est refusée, alors il est bloqué (attente passif).

Solutions avec arbitrage: Principe

Répéter
...
① Est-ce que je peux entrer en SC (je demande ceci à l'arbitre) ?
② Si NON alors je me bloque ...
<section critique>
Je relâche la SC (en le disant à l'arbitre), et la question se pose :
Est-ce qu'un processus est bloqué ?
Si OUI alors le réveiller (ou Ordonner un réveil)
Jusqu'à faux

- Dans le protocole de sortie de la SC, on fait réveiller le prochain thread (bloqué) qui attend accès à cette SC.

Solutions avec arbitrage: Principe

Répéter
...
① Est-ce que je peux entrer en SC (je demande ceci à l'arbitre) ?
② Si NON alors je me bloque ...
<section critique>
Je relâche la SC (en le disant à l'arbitre), et la question se pose :
Est-ce qu'un processus est bloqué ?
Si OUI alors le réveiller (ou Ordonner un réveil)
Jusqu'à faux

- Il faudrait exécuter la demande de permission et le blocage d'une façon indivisible, afin d'éviter les ordres de réveil perdus.

Solutions avec arbitrage: Principe

Répéter
...
① Est-ce que je peux entrer en SC (je demande ceci à l'arbitre) ?
② Si NON alors je me bloque ...
<section critique>
Je relâche la SC (en le disant à l'arbitre), et la question se pose :
Est-ce qu'un processus est bloqué ?
Si OUI alors le réveiller (ou Ordonner un réveil)
Jusqu'à faux

- Pareil dans le protocole de sortie, sinon on risque d'avoir plusieurs threads dans la SC.

Solutions en Java: Moniteur

- Le moniteur en Java est une solution qui vise la protection d'une donnée partagée en restreignant l'accès à un bloc du code modifiant cette donnée. Un moniteur se pose à l'aide du mot clé **synchronized**:

```
5   public class SynchronizedCounter {
6       private int c = 0;
7
8       public synchronized void increment() {
9           c++;
10      }
11
12      public synchronized void decrement() {
13          c--;
14      }
15
16      public synchronized int value() {
17          return c;
18      }
19  }
```

Solutions en Java: Moniteur

Synchronisation d'une méthode:

1. Il n'est pas possible d'avoir un entrelacement de deux appels à des méthodes synchronisées du même objet. Quand un thread X exécute une méthode synchronisée d'un objet O alors tout autre thread voulant accéder à des méthodes synchronisés de O sera bloqué jusqu'à ce que X a fini avec l'objet.

Solutions en Java: Moniteur

2 le protocole de sortie d'une méthode synchronisée invoque automatiquement une relation "Happens-before" avec tout appel à une méthode synchronisée du même objet. Ceci guarantie la consistance de la mémoire.

Solutions en Java: Moniteur

Synchronisation d'un bloc:

- Dans ce cas, on spécifie l'objet qu'on synchronise:

```
21    public void addName(String name) {
22        synchronized(this) {
23            lastName = name;
24            nameCount++;
25        }
26        Object nameList;
27        nameList.add(name);
28    }
```

- Cette méthode est utile pour éviter les problèmes causé par un appel dans le code synchronisé d'une méthode d'un autre objet.

Solutions en Java: Moniteur

Synchronisation d'une méthode statiques:

- Il est possible de synchroniser une méthode statique, mais elle est faite au niveau de l'objet de type Class, et non pas objet.
- 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())**{**...**}.

Solutions en Java: Locks

Interêt:

- Dans la synchronisation de bloc, le choix de l'objet sur lequel le moniteur est posé est important.
- En général, l'objet à passer à l'appel à synchroniser est l'objet à protéger et qui sera commun à tous les threads.
- Astuce: Créer un objet ad-hoc spécialement conçu pour la synchronisation (rejoint la notion de Lock):

```
// 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
  ... }
```

Solutions en Java: Locks

Exemple:

```
1  public class MsLunch {
2      private long c1 = 0;
3      private long c2 = 0;
4      private Object lock1 = new Object();
5      private Object lock2 = new Object();
6
7      public void inc1() {
8          synchronized(lock1) {
9              c1++;
0          }
1      }
2
3      public void inc2() {
4          synchronized(lock2) {
5              c2++;
6          }
7      }
8  }
```

Solutions en Java: Locks

Interface Lock:

- L'interface Lock fournit les primitives nécessaire pour manipuler différentes classes de 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 essayant un moniteur sur cette même ressource.

Solutions en Java: Locks

Interface Lock:

- **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.

Deadlock en Java:

- Quand un ou plusieurs threads sont bloqués en attente de la libération d'un lock par un autre thread.
- Ceci arrive quand plusieurs threads ont besoins de mêmes locks, en même temps mais dans un ordre différent.

Deadlock en Java: Exemple

```
4   class Friend {
5       private final String name;
6
7       public Friend(String name) {
8           this.name = name;
9       }
10
11      public String getName() {
12          return this.name;
13      }
14
15      public synchronized void bow(Friend bower) {
16
17          System.out.format("%s: %s"
18              + "  has bowed to me!%n",
19              this.name, bower.getName());
20          bower.bowBack(this);
21      }
22
23      public synchronized void bowBack(Friend bower) {
24          System.out.format("%s: %s"
25              + " has bowed back to me!%n",
26              this.name, bower.getName());
27      }
28
29  }
```

Deadlock en Java: Exemple

```
25      public static void main(String[] args) {
26          final Friend alphonse =
27              new Friend("Alphonse");
28          final Friend gaston =
29              new Friend("Gaston");
30          new Thread(new Runnable() {
31              public void run() { alphonse.bow(gaston); }
32          }).start();
33          new Thread(new Runnable() {
34              public void run() { gaston.bow(alphonse); }
35          }).start();
36      }
37  }
```

Deadlock en Java: Exemple

```
25      public static void main(String[] args) {
26          final Friend alphonse =
27              new Friend("Alphonse");
28          final Friend gaston =
29              new Friend("Gaston");
30          new Thread(new Runnable() {
31              public void run() { alphonse.bow(gaston); }
32          }).start();
33          new Thread(new Runnable() {
34              public void run() { gaston.bow(alphonse); }
35          }).start();
36      }
37  }
```

Résultat:

Console

```
Alphonse: Gaston  has bowed to me!
Gaston: Alphonse  has bowed to me!
```

Deadlock en Java: Solution des Lock Ordonné

- Établir un ordre sur l'acquisition du lock; Si un thread a besoin de plusieurs locks, il doit les acquérir dans l'ordre donné.
- Cette méthode est simple mais effective dans le cas où on connait en avance tous les locks dont le thread devrait obtenir.
- Le choix de l'ordre à déterminer.

Deadlock en Java: Exemple du Lock Ordonné

```
12        public void bow(Friend bower) {
13            Friend fst=this;
14            Friend scd=bower;
15            if(System.identityHashCode(bower)>System.identityHashCode(this))
16            {
17                fst=bower;
18                scd=this;
19            }
20            synchronized(fst) {
21                synchronized(scd) {
22                    System.out.format("%s: %s"
23                            + "  has bowed to me!%n",
24                            this.name, bower.getName());
25                        bower.bowBack(this);
26                }
27            }
28        }
```

Deadlock en Java: Exemple du Lock Ordonné

```
30      public void bowBack(Friend bower){
31          Friend fst=this;
32          Friend scd=bower;
33          if(System.identityHashCode(bower)>System.identityHashCode(this))
34          {
35              fst=bower;
36              scd=this;
37          }
38          synchronized(fst) {
39              synchronized(scd) {
40                  System.out.format("%s: %s"
41                          + " has bowed back to me!%n",
42                  this.name, bower.getName());
43              }
44          }
45      }
```

Deadlock en Java: Exemple du Lock Ordonné

```
25      public static void main(String[] args) {
26          final Friend alphonse =
27              new Friend("Alphonse");
28          final Friend gaston =
29              new Friend("Gaston");
30          new Thread(new Runnable() {
31              public void run() { alphonse.bow(gaston); }
32          }).start();
33          new Thread(new Runnable() {
34              public void run() { gaston.bow(alphonse); }
35          }).start();
36      }
37  }
```

Deadlock en Java: Exemple du Lock Ordonné

```
25      public static void main(String[] args) {
26          final Friend alphonse =
27              new Friend("Alphonse");
28          final Friend gaston =
29              new Friend("Gaston");
30          new Thread(new Runnable() {
31              public void run() { alphonse.bow(gaston); }
32          }).start();
33          new Thread(new Runnable() {
34              public void run() { gaston.bow(alphonse); }
35          }).start();
36      }
37  }
```

Résultat:

Console

```
Alphonse: Gaston  has bowed to me!
Gaston: Alphonse has bowed back to me!
Gaston: Alphonse  has bowed to me!
Alphonse: Gaston has bowed back to me!
```

<terminated> deadLockSolution [Java Application] C:\Program

Deadlock en Java: Autres solutions 1

1. **Lock Timeout :** Imposer une limite à la durée qu'un thread pourrait rester en attente d'un lock.

```
boolean tryLock(long time,
                TimeUnit unit)
                throws InterruptedException
```

2. **DeadLock Detection :** Détecter l'existence des deadlocks, en gardant trace dans une structure de données les locks demandés et/ou obtenus par chaque thread. A chaque fois une demande d'acquisition d'un lock est refusée, la structure de données est parcourue afin de vérifier si il s'agit d'un deadlock.

Deadlock en Java: Autres solutions 2

Remarque: Ces deux solutions détectent les deadlock mais ne les résolvent pas. Pour ceci:

- Le thread pourrait faire un backup, libérer ses locks acquis puis attendre une durée aléatoire avant de réessayer.
- Ou bien assigner des priorités aux threads (aléatoirement), afin de faire avancer l'exécution.

Sémaphores en Java:

- Une extension de la notion de lock à plusieurs permission.
- Chaque sémaphore a un nombre de tokens (permissions) borné, défini au moment de son initialisation.
- Avant d'accéder à la SC, chaque thread doit d'abord acquérir un token du sémaphore.
- Si un thread ne trouve pas de jeton disponible, il reste bloqué sur l'acquisition jusqu'à ce qu'un autre thread relâche un jeton.

Sémaphores en Java: Exemple

```
import java.util.concurrent.Semaphore;   1

class Pool {
    private static final int MAX_AVAILABLE = 100;

    private final Semaphore available =
                new Semaphore(MAX_AVAILABLE, true);   2

    public Object getItem() throws InterruptedException {
                    available.acquire();   3
                    return getNextAvailableItem();
                    }
    public void putItem(Object x) {
        if (markAsUnused(x))
            available.release(); 4
        }
```

Sémaphores en Java: Sémaphore Vs Locks

- Un sémaphore binaire (Un seul jeton disponible) pourrait jouer le rôle d'un lock classique. Mais....

Sémaphores en Java: Sémaphore Vs Locks

- Un sémaphore binaire (Un seul jeton disponible) pourrait jouer le rôle d'un lock classique. Mais....
- Il n'y a aucune contrainte sur l'identité des objets et des threads qui acquièrent ou relâchent un jeton (il est conseillé, par contre, de l'acquérir et le relâcher dans la même classe). Donc...

Sémaphores en Java: Sémaphore Vs Locks

- Un sémaphore binaire (Un seul jeton disponible) pourrait jouer le rôle d'un lock classique. Mais....
- Il n'y a aucune contrainte sur l'identité des objets et des threads qui acquièrent ou relâchent un jeton (il est conseillé, par contre, de l'acquérir et le relâcher dans la même classe). Donc...
- Un sémaphore binaire peut être relâché par un thread différent, ce qui ne permet pas toujours de faire un lock. Et ceci pourrait être utile afin d'éviter les deadlocks.

Timers en Java:

- Classe permettant des threads de planifier des tâches qui seront exécuter dans le futur une seule fois ou périodiquement.

Timers en Java:

- Classe permettant des threads de planifier des tâches qui seront exécuter dans le futur une seule fois ou périodiquement.
- Un thread par objet Timer.

Timers en Java:

- Classe permettant des threads de planifier des tâches qui seront exécuter dans le futur une seule fois ou périodiquement.
- Un thread par objet Timer.
- Ce thread ne s'execute pas par défaut comme **Daemon**.

Timers en Java:

- Classe permettant des threads de planifier des tâches qui seront exécuter dans le futur une seule fois ou périodiquement.
- Un thread par objet Timer.
- Ce thread ne s'execute pas par défaut comme **Daemon**.
- On peut l'arrêter à n'importe quelle moment en appelant la méthode **cancel()**.

Timers en Java: Exemple

```
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());
                }
    }
```

Timers en Java: Exemple

```
public class TimeExample {
    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!");
    }
}
```