Programação Concorrente em Java

Aula 12

Deadlock

Introdução

- Um situação de **deadlock** acontece quando duas ou mais threads estão bloqueadas esperando obter as travas de outras threads.
- Ocorre quando múltiplas threads necessitam das mesmas travas, ao mesmo tempo mas as obtem em ordem diferente.

```
Thread 1  locks A, waits for B
Thread 2  locks B, waits for A
```

Exemplo 1

```
public class TreeNode {

  TreeNode parent   = null;
  List children = new ArrayList();
  public synchronized void addChild(TreeNode child){
    if(!this.children.contains(child)) {
      this.children.add(child);
      child.setParentOnly(this);
    }
  }

  public synchronized void addChildOnly(TreeNode child){
    if(!this.children.contains(child){
      this.children.add(child);
    }
  }

...
```

```
...
public synchronized void setParent(TreeNode parent){
    this.parent = parent;
    parent.addChildOnly(this);
  }
  public synchronized void setParentOnly(TreeNode parent){
    this.parent = parent;
  }
}
```

Exemplo 2

```
public class Deadlock {
    static class Friend {
        private final String name;
        public Friend(String name) {
            this.name = name;
        }
        public String getName() {
            return this.name;
        }
        public synchronized void bow(Friend bower) {
            System.out.format("%s: %s"
                + "  has bowed to me!%n",
                this.name, bower.getName());
            bower.bowBack(this);
        }
        public synchronized void bowBack(Friend bower) {
            System.out.format("%s: %s"
                + " has bowed back to me!%n",
                this.name, bower.getName());
        }
    }
...
```

```
 ...
    public static void main(String[] args) {
        final Friend alphonse =
            new Friend("Alphonse");
        final Friend gaston =
            new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
            public void run() { gaston.bow(alphonse); }
        }).start();
    }
}
```

Exemplo

- If a thread (1) calls the parent.addChild(child) method at the same time as another thread (2) calls the child.setParent(parent) method, on the same parent and child instances, a deadlock can occur. Here is some pseudo code that illustrates this:

```
Thread 1: parent.addChild(child); //locks parent
          --> child.setParentOnly(parent);
Thread 2: child.setParent(parent); //locks child
          --> parent.addChildOnly()
```

Exemplo

- First thread 1 calls parent.addChild(child). Since addChild() is synchronized thread 1 effectively locks the parent object for access from other treads.
- Then thread 2 calls child.setParent(parent). Since setParent() is synchronized thread 2 effectively locks the child object for acces from other threads.
- Now both child and parent objects are locked by two different threads. Next thread 1 tries to call child.setParentOnly() method, but the child object is locked by thread 2, so the method call just blocks. Thread 2 also tries to call parent.addChildOnly() but the parent object is locked by thread 1, causing thread 2 to block on that method call. Now both threads are blocked waiting to obtain locks the other thread holds.

Exemplo

- Note: The two threads must call parent.addChild(child) and child.setParent(parent) at the same time as described above, and on the same two parent and child instances for a deadlock to occur. The code above may execute fine for a long time until all of a sudden it deadlocks.
- The threads really need to take the locks *at the same time*. For instance, if thread 1 is a bit ahead of thread2, and thus locks both A and B, then thread 2 will be blocked already when trying to lock B. Then no deadlock occurs. Since thread scheduling often is unpredictable there is no way to predict *when* a deadlock occurs. Only that it *can* occur.

Casos mais complicados

- Quando inclue-se mais de uma thread...

```
Thread 1  locks A, waits for B
Thread 2  locks B, waits for C
Thread 3  locks C, waits for D
Thread 4  locks D, waits for A
```

- Thread 1 waits for thread 2, thread 2 waits for thread 3, thread 3 waits for thread 4, and thread 4 waits for thread 1.

Deadlocks em Databases

- Um situação complicada que pode ocorrer em bancos de dados.
- Uma transação pode consistir de vários requesições de update em SQL.
- Quando um registro é atualizado durante uma transação, esse registro é “locked” para outras transações de updates.

```
Transaction 1, request 1, locks record 1 for update
Transaction 2, request 1, locks record 2 for update
Transaction 1, request 2, tries to lock record 2 for update.
Transaction 2, request 2, tries to lock record 1 for update.
```

Deadlocks

Prevenção de Deadlocks

Técnicas

- Algumas técnicas podem ser usadas para previnir deadlocks, dentre elas:
  - Ordenação dos locks (Lock ordering);
  - Lock timeout;
  - Detecção de deadlock (antes que ele ocorra).

Lock ordering

- Deadlock ocorre quando múltiplas threads precisam dos mesmos locks mas os obtem em ordens diferentes.
- Se o programador tiver certeza que todas as locks serão obtidas na mesma ordem por qualquer thread, então deadlocks nunca irão ocorrer.

Lock ordering

```
Thread 1:
  lock A
  lock B
Thread 2:
   wait for A
   lock C (when A locked)
Thread 3:
   wait for A
   wait for B
   wait for C
```

- If a thread, like Thread 3, needs several locks, it must take them in the decided order. It cannot take a lock later in the sequence until it has obtained the earlier locks.
- For instance, neither Thread 2 or Thread 3 can lock C until they have locked A first. Since Thread 1 holds lock A, Thread 2 and 3 must first wait until lock A is unlocked. Then they must succeed in locking A, before they can attempt to lock B or C.

Lock timeout

- Outra forma de impedir deadlocks é colocar um timeout em uma lock, impedindo que uma thread fique muito tempo esperando.
- Se uma thread falhar em conseguir todas as locks, ela irá desistir, fazer um backup do seu estado atual, liberar as outras locks, dormir por um tempo e tentar novamente no futuro.
- O tempo de dormida deve ser randômico, para que todas as threads envolvidas tenham chance em pegar as travas que precisam.

Lock timeout

```
Thread 1 locks A
Thread 2 locks B

Thread 1 attempts to lock B but is blocked
Thread 2 attempts to lock A but is blocked

Thread 1's lock attempt on B times out
Thread 1 backs up and releases A as well
Thread 1 waits randomly (e.g. 257 millis) before retrying.

Thread 2's lock attempt on A times out
Thread 2 backs up and releases B as well
Thread 2 waits randomly (e.g. 43 millis) before retrying.
```

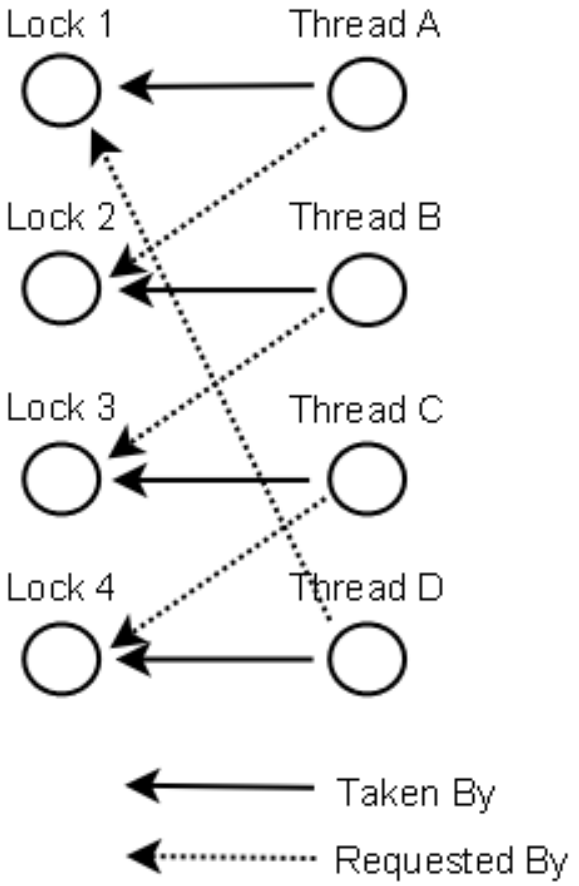
Detecção de Deadlock

- É uma abordagem pesada, onde nem o timeout e nem a ordenação são possíveis.
- Toda vez que uma thread **pega** um lock, isso é anotado em uma estrutura de dados (mapa, grafo, lista, etc.).
- Além disso, toda vez que uma thread requisita um lock, isso também é anotado na mesma estrutura de dados.

Detecção de Deadlock

- When a thread requests a lock but the request is denied, the thread can traverse the lock graph to check for deadlocks. For instance, if a Thread A requests lock 7, but lock 7 is held by Thread B, then Thread A can check if Thread B has requested any of the locks Thread A holds (if any). If Thread B has requested so, a deadlock has occurred (Thread A having taken lock 1, requesting lock 7, Thread B having taken lock 7, requesting lock 1).
- Of course a deadlock scenario may be a lot more complicated than two threads holding each others locks. Thread A may wait for Thread B, Thread B waits for Thread C, Thread C waits for Thread D, and Thread D waits for Thread A. In order for Thread A to detect a deadlock it must transitively examine all requested locks by Thread B. From Thread B's requested locks Thread A will get to Thread C, and then to Thread D, from which it finds one of the locks Thread A itself is holding. Then it knows a deadlock has occurred.

Detecção de Deadlock



Detecção de Deadlock

- Quando o deadlock é detectado, o quê a thread faz?
  - One possible action is to release all locks, backup, wait a random amount of time and then retry. This is similar to the simpler lock timeout mechanism except threads only backup when a deadlock has actually occurred. Not just because their lock requests timed out. However, if a lot of threads are competing for the same locks they may repeatedly end up in a deadlock even if they back up and wait.
  - A better option is to determine or assign a priority of the threads so that only one (or a few) thread backs up. The rest of the threads continue taking the locks they need as if no deadlock had occurred. If the priority assigned to the threads is fixed, the same threads will always be given higher priority. To avoid this you may assign the priority randomly whenever a deadlock is detected.

Referência

- http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html