Deadlocks in Java and how to avoid them.

Introduction

Writing multithreaded code is considered more difficult and demanding than regular programming, and not without a reason. In addition to regular programming tasks, a creator of a multithreaded software have to worry about the race conditions over shared data. The potential races may harm the computations and leave the data in inconsistent state, so they should be discovered and fixed, which is most often done using thread synchronization. The latter is a great tool, but if not implemented carefully, may result in deadlocks in the application – the nasty bugs, very hard to discover, reproduce and fix.

This article introduces the two basic approaches to implementing a deadlock-free synchronization. The intended audience is assumed to be familiar with the main synchronization primitives, to have some experience in multithreaded programming and to be eager to advance their skills in this area.

1. The mechanics of deadlocks.

Let's recollect in detail on how the deadlocks work. Consider the following code:

```java
public class MyDeadlock{
   static final Object lock1=new Object();
   static final Object lock2=new Object();

   static int variable;

   static void increment(){
      synchronized(lock1){
         synchronized(lock2){
            variable++;
         }
      }
   }

   static void decrement(){
      synchronized(lock2){
         synchronized(lock1){
            variable--;
         }
      }
   }

   public static void main(String[] args) {
      int N=50_000;

      new Thread(() -> {
         for(int i=N; i-->0;) increment();
         System.out.println("increments done");
      }).start();

      new Thread(() -> {
         for(int i=N; i-->0;) decrement();
         System.out.println("decrements done");
      }).start();
   }
}
```

In almost every attempt to run this class the program would hang while the thread dump (using CTRL-Break key or jstack.exe) would inform us of a found deadlock. Let's consider in detail what happens to the threads in this example.

Both *increment()* and *decrement()* consist of 5 steps:

Table 1

#	increment()	decrement()
1	Acquire lock1	Acquire lock2
2	Acquire lock2	Acquire lock1
3	Perform increment	Perform decrement
4	Release lock2	Release lock1
5	Release lock1	Release lock2

Obviously, the steps 1 and 2 are passable only if the corresponding locks are free, otherwise the thread would block until the lock's release.

Suppose there are two threads executing the above two methods in parallel. Each thread's steps will be performed in the normal order, but the steps of one thread will be randomly interleaved with the steps of another thread. The randomness comes from unpredictable delays imposed by the system thread scheduler. The possible interleaving patterns are quite numerous (to be exact, there are 252 of them), and they all fall into the two groups. The first group is where the sequence begins with a single thread acquiring both locks (see Table 2). All cases in this group result in normal execution.

Table 2

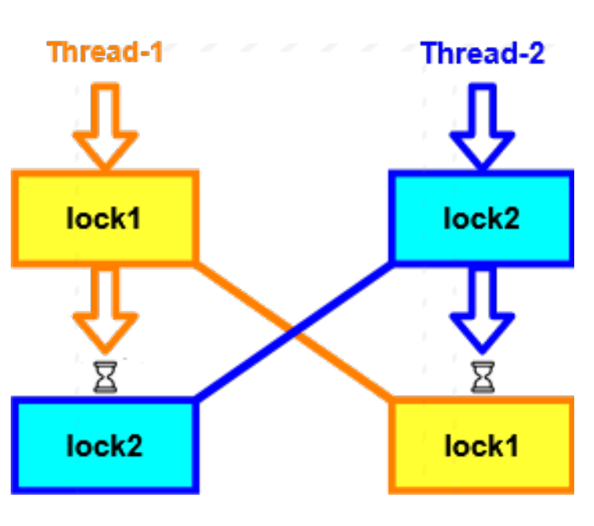
Group 1 pattern, no deadlock		
Thread-1	Thread-2	Result
1: Acquire lock1		lock1 busy
2: Acquire lock2		lock2 busy
	1: Acquire lock2	wait for lock2 release
3: Increment	Waiting at lock2	
4: Release lock2	Intercept lock2	lock2 changed owner
	2: Acquire lock1	wait for lock1 release
5: Release lock1	Intercept lock1	lock1 changed owner
	3: Decrement	
	4: Release lock1	lock1 free
	5: Release lock2	lock2 free

Table 3

Group 2 pattern, deadlock		
Thread-1	Thread-2	Result
1: Acquire lock1		lock1 busy
	1: Acquire lock2	Lock2 busy
2: Acquire lock2		wait for lock2 release
	2: Acquire lock1	wait for lock1 release
Waiting at lock2	Waiting at lock1	
…	…	

In the second group the sequence begins with both threads having acquired a lock (see Table 3). All cases in this group result in the situation where the first thread waits for the lock that is owned by the second thread, and the second thread waits for the lock that is owned by the first thread, so the both threads cannot progress any further:

Figure 1



This is what is called a deadlock. Let's outline its most obvious features:

- It requires at least two threads and two locks per thread to happen
- It's probabilistic: it happens only at certain combinations of the thread timings
- It depends on the locking order

Now we are ready to proceed to the first strategy for writing deadlock-free code. But first let's introduce the toy project that we will use as a model for implementing synchronization schemes.

2. The toy project

Let's write a simple bank application, capable of opening and closing accounts, depositing, withdrawing and transferring money between accounts. The bank will operate in the indivisible currency named 'credits'. And, due to government regulations, no account balance may exceed 2^20 credits. The license also doesn't allow making loans, so negative account balances aren't allowed too.

Because the credits are indivisible and the maximum balances are limited, it is correct to represent the money by a primitive long type. To keep it simple, we will represent the accounts by their identifier of type Long. The methods should throw an appropriate exception in the cases when the requested account doesn't exist, or the attempted spending exceeds the account balance, or the attempted top-up causes a limit overflow. That is, our bank should implement the following interface:

```java
public interface ToyBank {
   /**
    * the limit for every account's balance
    */
   public long MAX_BALANCE = 1L<<20;

   /**
    * Create account
    */
   public Long createAccount(long initialDeposit) throws BalanceOverflow;

   /**
    * Delete account, return the balance
    */
   public long deleteAccount(Long accountId) throws NotFound;

   /**
    * Deposit money.
    * The amount should match the account's limit.
    */
   public void deposit(Long accountId, long amount) throws NotFound, BalanceOverflow;

   /**
    * Withraw money.
    * The amount should match the account's balance.
    */
   public void withdraw(Long accountId, long amount) throws NotFound, InsufficientBalance;

   /**
    * Transfer money between accounts.
    * The amount must match the source's balance and the destination's limit.
    */
   public void transfer(Long srcId, Long dstId, long amount) throws NotFound, InsufficientBalance, BalanceOverflow;

   /**
    * @return account's total
    */
   public long getBalance(Long accountId) throws NotFound;

   /**
    * @return sum of all accounts
    */
   public long totalValue();
}
```

Of course, the bank should be thread-safe, but, for the sake of simplicity, we don't demand it to be persistent.

We are going to solve this task in two steps. In the first step we will create a basic non-thread-safe implementation. In the second we will add the synchronization.

Let's start with design of the data structures, keeping them as simple as possible. Let's represent the accounts database by a HashMap which maps the account id of type *Long* to the account balance value implemented as a primitive long array of length 1. The identifiers will be created using a simple sequence generator implemented as a primitive long. This all seems to be enough for the bookkeeping:

```java
   private final Map<Long,long[]> db=new HashMap<>();
   private long idGenerator = 1;
```

The implementations of the interface methods are then quite straightforward, e.g.:

```java
   public Long createAccount(long initialDeposit) throws BalanceOverflow{
      if(initialDeposit < 0) throw new IllegalArgumentException("negative initialDeposit: " + initialDeposit);
      if(initialDeposit > MAX_BALANCE) throw new BalanceOverflow(null, initialDeposit, 0);

      Long id = idGenerator++;
      db.put(id, new long[]{initialDeposit});

      return id;
   }


   public void transfer(Long srcId, Long dstId, long amount) throws NotFound, InsufficientBalance, BalanceOverflow{
      if(amount < 0) throw new IllegalArgumentException("negative amount: " + amount);

      long[] srcValue = db.get(srcId);
      if(srcValue == null) throw new NotFound(srcId);
      if(amount > srcValue[0]) throw new InsufficientBalance(srcId, amount, srcValue[0]);

      long[] dstValue = db.get(dstId);
      if(dstValue == null) throw new NotFound(dstId);
      if(amount > MAX_BALANCE-dstValue[0]) throw new BalanceOverflow(dstId, amount, dstValue[0]);

      srcValue[0] -= amount;
      dstValue[0] += amount;
   }
```

The full code is here: ToyBankBase.java. Now as the implementation step is over, let's proceed to the synchronization step.

The first synchronization strategy we will try is the coarse-grained synchronization.

3. Coarse-grained synchronization

The idea is pretty obvious. As was noted in the Section 1, deadlocks may happen only if each of the concurrent transactions holds at least two locks. So if we ensure that no transaction holds more than one lock at once, we would eliminate the physical possibility of deadlocks.
But how much locks can we use and which lock should be assigned to which transaction? The most straightforward answer is to use a single lock for all the transactions in the system. This approach is correct but not perfect, let's see why. Consider the following class:

```java
public class Groups {
   int A, B, C, D;

   synchronized void transaction1(){
      A += B<0? 1: -1;
      B += A>0? 1: -1;
   }

   synchronized void transaction2(){
      C += D<0? 1: -1;
      D += C>0? 1: -1;
   }
}
```

Note, that the variables form the two independent groups, {A,B} and {C,D}. If we protect each group by its own lock instead of using the global one the transactions wouldn't block each other while remaining atomic. The two transactions then could be executed concurrently, which is beneficial for performance:

```java
   void transaction1(){
      synchronized(lock1){
         A += B<0? 1: -1;
         B += A>0? 1: -1;
      }
   }

   synchronized void transaction2(){
      synchronized(lock2){
         C += D<0? 1: -1;
         D += C>0? 1: -1;
      }
   }
```

This hints us that the optimal synchronization should use a dedicated lock per each such group of variables. In order to formulate it more accurately let's define a few terms:

- The ***data element*** is a data structure or a part of it that has a fixed address between the transactions. This may be an instance or static field, or an element of array. A group of data elements may be treated as a single composite data element.
- Two data elements are ***transactionally connected*** if there exists a transaction that accesses both of them. This property is transitive, i.e. if A is connected to B, B is connected to C, then A is connected to C.
- The group of data elements is ***transactionally closed*** if all its members are mutually connected and have no connected elements outside of this group.

Then the rule for the optimal coarse-grained synchronization reads:

- ***Each transactionally closed group of data elements should be protected by a single dedicated lock***

To illustrate this rule, let's turn back to our ToyBank example. In order to find out the closed groups of data elements let's write down the table with methods and the accessed data:

Table 4

Method	Accessed data elements			
	idGenerator	db structure	i-th account	j-th account
createAccount()	+	+	+	
deleteAccount()	-	+	+	
deposit()	-	+	+	
withdraw()	-	+	+	
transfer()	-	+	+	+

We see that all the data elements are mutually connected via one or more methods, so they all form a single transactionally closed group. Therefore all the transactions should be protected by a single lock, which we can achieved by just making all the methods *synchronized*.

The resulting class is here: ToyBankCoarse.java.

The main advantage of the coarse-grained synchronization is the simplicity. The synchronization code is small and straightforward, leaving almost no space for developer's mistakes.

The main drawback, of course, is its unfriendliness to a parallel execution, because any transaction blocks the rest ones in its group. And the effect of this is twofold. First, the load isn't distributed to the multiple processor cores, which is a waste of resources. Second, the higher is the level of contention over a lock, the more CPU cycles it takes to acquire it, which leads to a significant drop in the overall performance as the number of concurrent requests grows. So the coarse-grained synchronization is not the right way to cope with highly concurrent loads, in such situations we need to look for another approach.

4. Fine-grained synchronization with a lock ordering

The key to improving parallelism is a synchronization with a much finer granularity. Ideally, each concurrently executed transaction should be synchronized on a separate lock. Besides that, the potential locking scheme must ensure that no data element is accessed by more than one transaction at a time. Keeping this all in mind, we come to the following design idea:

1. ***Let each data element to have a corresponding dedicated lock***
2. ***Execute each transaction within a protective block formed by acquiring all the locks that correspond to all the involved data elements***

The following code illustrates this idea:

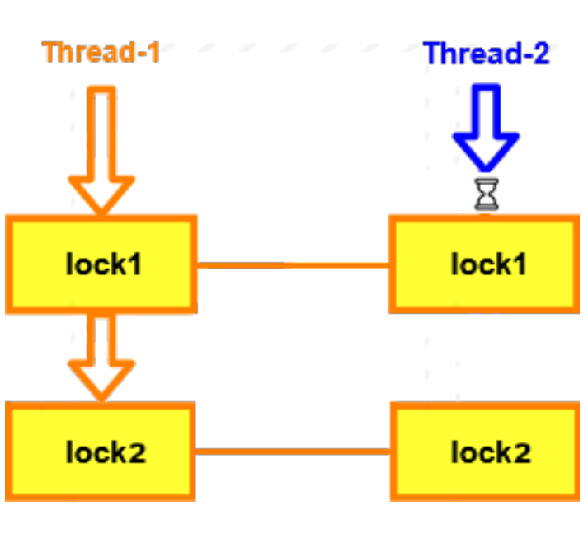
```java
public class FineGrainedLocking {
   int[] data;
   final Object[] locks;

   void transfer(int from, int to, long amount){
      Object lock1 = locks[from];
      Object lock2 = locks[to];

      synchronized (lock1) {
         synchronized (lock2) {
            data[from] -= amount;
            data[to] += amount;
         }
      }
   }
}
```

Due to such protection for any data element all the transactions involving this element would be atomic and strictly sequential. As a consequence, any transaction in the system preserves the data integrity. At the same time, the transactions that work on different data elements are mutually non-blocking, so the system supports certain level of parallelism, which comes close to 100% for large enough number of data elements.

So far the approach looks quite promising, except one important detail. Since we have introduced a synchronization on multiple locks, we now face the possibility of deadlocks. But, fortunately, with minimal changes to the locking scheme we can make it deadlock-free.

Let's look again at the picture in the Section 1. It is obvious that if the both threads acquire the locks in the same order, the deadlocks would be impossible, because the thread that took the first lock would control both of them. Another thread would wait on the first lock until the winning thread release both the locks:

Figure 2



In a sense, multiple locks would work like a single one. Intuitively it is clear that this should work for any combinations of threads and locks. That is, if in any transaction in the system the same locks are acquired in the same order, we expect this system to be deadlock-free. In fact, this assumption needs to be supplemented by a transitive rule to become absolutely correct. That is, if there exist locking sequences A->B and B ->C, then the allowed order for the locks A and C is A->C, not the other way round. The more compact formulation of the lock ordering rule is as follows (it supplements the two design rules above):

3. ***The locks in the rule 2 should be acquired according to a predefined global order of precedence***

It can be proved, that the rule 3 spares us from deadlocks. As an illustration, let's apply it to the *FineGrainedLocking* class above. Let the order of precedence to be determined by the account index, the smaller index coming first. The only change will be the lock ordering step in the *transfer()* method:

```java
   void transfer(int from, int to, long amount){
      Object lock1 = locks[Math.min(from, to)];
      Object lock2 = locks[Math.max(from, to)];

      synchronized (lock1) {
         synchronized (lock2) {
            data[from] -= amount;
            data[to] += amount;
         }
      }
   }
```

The three rules above make up all the pieces needed to implement a concurrent yet thread-safe system. Let's apply them to the toy bank project.

Let's plan it out, starting with the non-synchronized *ToyBankBase*.

First, we need to provide the locks for all the data elements. The lock for the database and id generator will be discussed a bit later. Then, we need a lock for each account. We may avoid creating extra objects with a clever trick: let's just use the account value objects as their own locks.

Next, we'll need to define a global locking order. Let the database lock always come first, and the account locks be ordered in accordance with the account identifiers, the lower one coming first. Therefore, each method will have the following structure:

1. acquire the database lock
2. look up the account objects
3. sort them by id and synchronize on them all
4. within the synchronized block perform the computation
5. release the locks

At this point we face a problem. As we are going to synchronize all the methods on the database lock, they are going to become mutually blocking, making the implementation non-concurrent.

To work around this problem we have to take an even finer-grained look at the access table from the Section 3 (Table 4). Let's reproduce it, this time taking into account the type of access (read or write). For brevity's sake we consider the id generator to be the part of the database structure.

Table 5

Method	Affected data elements		
	Database	i-th account	j-th account
createAccount()	RW	RW	
deleteAccount()	RW	RW	
deposit()	R	RW	
withdraw()	R	RW	
transfer()	R	RW	RW

We see that there are methods of the two kinds, the ones that write to the database and the ones that only read it. The idea is to split the database lock into the read and write ones using the Java Lock API, then use the write lock in the first two methods and the read one in the rest three. As read locks are non-exclusive, the last three methods would be mutually non-blocking. And, as these methods are supposed to be the most often called ones, the resulting implementation may be considered the almost-concurrent. The resulting code can be found here: ToyBankOrdered.java.

Another method of solving the database lock problem is to get rid of it whatsoever, switching to the use of *ConcurrentHashMap* for the database and *AtomicLong* for the id generator. In this case we would have to take special measures to make account deletions globally atomic. The latter could be achieved by marking the deleted accounts by a negative value and checking for this mark in all the transactions. In this case the thread that could be locked on the account while it was being deleted would know about the occurred deletion and would cancel the transaction. Such implementation is fully concurrent, at the price of higher complexity and loss of the consistent view of the database in the *totalValue()* method. The code of this variant is here: ToyBankConcurrent.java.

Finally, we have to discuss the scalability aspect of the fine-grained synchronization. Keeping a separate lock for each data element can be an obstacle to the scalability of an application, because the locks in this design are live objects, and live objects in java are not free performance-wise. First, they take up memory. Second, large numbers of live objects slow down the garbage collection, which results in the decreased overall throughput and responsiveness. Fortunately, keeping large numbers of locks can be avoided using lock pooling.

4.1 Fine-grained synchronization with lock pooling

In the previous section we interpreted the design rule 1 as requiring to have a separate lock for each data element, that is, for N data elements we had to provide N lock objects. But the rule could be interpreted differently, without breaking the design. Instead, we can map all the data elements into the fixed set of M locks, where M significantly exceeds the number of threads yet is small enough to cause no scalability problems. Except for massively multicore systems, the optimal value of M would be in the range of a couple of hundreds. To ensure the global ordering of locks, the following requirements should be met:

- The mapping should remain constant during the application run time
- The ordering step should be performed after the mapping step, not the other way around

The following code illustrates the above considerations:

```java
public class FineGrainedScalableLocking {
   static final int POOL_SIZE = 1<<8;
   static final int POOL_MASK = POOL_SIZE-1;

   final long[] data;
   final Object[] lockPool = new Object[POOL_SIZE];

   void transfer(int from, int to, long amount){
      Object lock1 = lockPool[Math.min(from & POOL_MASK, to & POOL_MASK)];
      Object lock2 = lockPool[Math.max(from & POOL_MASK, to & POOL_MASK)];

      synchronized (lock1) {
         synchronized (lock2) {
            data[from] -= amount;
            data[to] += amount;
         }
      }
   }
}
```

Note that the ordering in the method *transfer()* is performed over the mapped indices, not the original ones.

The fine-grained synchronization with its variants therefore makes an almost ideal solution, which ensures the integrity of data while providing the desired level of concurrency and scalability. The only drawback is its complexity. To implement it one have to delve into the business logic and properly mix in the synchronization code, which requires certain skills.

5. Limitations of applicability. Interaction with hidden locks.

Even though the synchronization techniques described in this article do deliver what they promise, they are not a silver bullet.

First, there may be situations where the intended synchronization strategy just doesn't fit into the application structure, and you don't have the right to change the latter.

Another potential source of problems is interaction with hidden locks in the environment, which may be an OS, a framework, or even a library. The above sections assume that the discussed code stands completely alone, but in reality we normally write code that closely interact with the environment via external calls and callbacks. But the environment may already be using its own locks which may interfere with our code in an unexpected way.

Let's consider two examples:

```java
public class MyHiddenDeadlock {
   private final ConcurrentHashMap<Integer, Integer> map = new ConcurrentHashMap<>();
   {
      map.put(1, 0);
      map.put(2, 0);
   }

   public void method1(){
      map.compute(1, (key,value)->{
         map.put(2, 1);
         return value;
      });
   }

   public void method2(){
      map.compute(2, (key,value)->{
         map.put(1, 1);
         return value;
      });
   }
}

public class AnotherHiddenDeadlock {
   private Hashtable<String, Long> db = new Hashtable<>();
   private long version;

   public void put(String key, long value){
      updateVersion(key);
      db.put(key, value);
   }

   public void increment(String key){
      db.computeIfPresent(key, (k,v)->{
         updateVersion(k);
         return v+1;
      });
   }

   private synchronized void updateVersion(String key){
      db.put(key+".version", version++);
   }
}
```

Judging by the number of used locks and according to Section 1, both classes should be deadlock-safe, but in fact they both are deadlock-prone. The first class contains no synchronization code at all, so its deadlocking potential results from the interaction with the *ConcurrentHashMap*. The latter contains multiple independently synchronized bins, and the given code potentially puts their locks into entanglement. The second class contains only one explicit lock, but there is another hidden one in the Hashtable object, and the two also become entangled in the given code.

So it may be concluded that implementing any synchronization scheme described in the previous sections wouldn't save us from accidental interference with hidden locks in the environment. And we would know about such an accident only in the application testing phase, or even worse, in production. Unfortunately, there is no ready-made general solution to this problem, only a few recommendations:

1. Be suspicious about hidden locks in the library/framework/OS
2. If possible, use only callbacks or only API calls, not both at once
3. Avoid making API calls in callbacks

And no matter what, whether you are designing a new software or fixing the existing one, you should be very critical of your synchronization schemes and should permanently analyze them for an appearance of potential deadlocks. The systematic method for such analysis will be introduced in the next article.

6. List of code samples

All sources can be found at https://github.com/samokhodkin/articles/blob/master/Deadlocks/src/deadlocks/

Deadlock examples:

- MyDeadlock.java – Basic deadlock example
- MyHiddenDeadlock.java - Deadlock caused by hidden locks
- AnotherHiddenDeadlock.java - Deadlock caused by an explicit lock and a hidden one

Coarse grained locking examples:

- Groups.java - Example of transactionally independent groups of variables

Fine-grained locking examples

- FineGrainedLocking.java - Example of fine-grained locking with lock ordering
- FineGrainedScalableLocking.java - Example of fine-grained locking with lock pooling

Toy Bank API:

- ToyBank.java – The Toy Bank interface
- NotFound.java – Account not found exception
- InsufficientBalance.java – Insufficient balance for a transaction
- BalanceOverflow.java – A balance is going to exceed the allowed limit

Toy Bank implementations:

- ToyBankBase.java — Basic non-thread-safe implementation
- ToyBankCoarse.java — Implementation using coarse-grained synchronization
- ToyBankOrdered.java — Partially concurrent implementation using fine-grained synchronization
- ToyBankConcurrent.java — Fully concurrent implementation using fine-grained synchronization
- ToyBankTest.java – Test for thread-safety and performance

To be continued.