42

CONSISTENT LOCK ORDER

DESCRIPTION

During the discussion of the Monitor and the Critical Section patterns earlier, we have seen that when the synchronized keyword is used to ensure single-threaded execution of a code block, a thread needs to wait while trying to acquire the lock associated with the specified object. Consider a scenario where two threads hold locks on two different objects and each one is waiting for a lock on the object that is locked by the other thread. Both threads will be waiting forever and are said to be in a state of deadlock. In terms of implementation, this type of situation most often occurs due to an inconsistent order of locking objects. Let us consider the code segment in Listing 42.1 to illustrate how inconsistent locking in a multithreaded environment can cause a deadlock.

Consider a scenario where:

- Two threads, A and B, simultaneously invoke methods — `Method_A` and `Method_B` — respectively on the same `SomeClass` object.
- Thread A acquires a lock on `objectA` and Thread B acquires a lock on `objectB` at the same time. At this point, each of the threads waits for a lock on the object locked by the other thread and this puts Thread A and Thread B in a deadlocked condition.

To address such deadlock issues, the Consistent Lock Order pattern recommends designing an object locking order to be followed consistently across an application. Simply following an object locking order consistently across the application (where objects of a particular class are to be locked before locking other class instances) can eliminate the deadlock problem associated with the example code block. In other words, by ensuring that objects are locked in a consistent order all across the application, the problem of deadlocks can be addressed.

The example code block in Listing 42.1 can be modified so that `ClassA` objects are locked prior to locking `ClassB` objects.

Listing 42.1 Class with Inconsistent Locking Order

```
public class SomeClass {
  private ClassA objectA;
  private ClassB objectB;
  public SomeClass() {
    objectA = new ClassA();
    objectB = new ClassB();
  }
  public void Method_A() {
    synchronized (objectA) {
      synchronized (objectB) {
        process_A();
      }
    }
  }
  public void Method_B() {
    synchronized (objectB) {
      synchronized (objectA) {
        process_B();
      }
    }
  }
  private void process_A() {
    //
  }
  private void process_B() {
    //
  }
}
class ClassA {
}
class ClassB {
}
```

```
public void Method_A() {
  synchronized (objectA) {
    synchronized (objectB) {
      process_A();
    }
  }
}
public void Method_B() {
  synchronized (objectA) {
    synchronized (objectB) {
      process_B();
    }
  }
}
```

This type of object locking order based on the class type does not work when the objects to be locked are instances of the same class. A more sophisticated algorithm may be needed to decide the object locking order. The following example illustrates one such mechanism.

EXAMPLE

Let us build a utility class that offers the functionality to move the contents of a directory to a different directory in the file system.

Let us create a class `Directory`, instances of which can be used to represent directories in the file system.

```
public class Directory {
  private String name;
  public Directory(String n) {
    name = n;
  }
}
```

The utility class `FileSysUtil` in its simplest form can be designed with a method to move the contents between directories.

```
public class FileSysUtil {
  public void moveContents(Directory src, Directory dest) {
    synchronized (src) {
      synchronized (dest) {
        System.out.println("Contents Moved Successfully");
      }
    }
  }
}
```

To move the contents of a directory to another, a client object or thread needs to:

1. Create `Directory` objects corresponding to the source and destination directories.
2. Invoke the `moveContents` method by passing both the `Directory` objects created in Step 1.

As part of its implementation of the `moveContents` method, the `FileSysUtil` locks the `Directory` objects representing the source and destination directories in sequence before actually moving the directory contents. This is to prevent threads from changing or deleting the source or destination directories while the current thread is in the process of moving the source directory contents to the destination directory. For simplicity, the example application displays an appropriate message instead of actually moving the source directory contents.

Let us suppose that there exist two directories — `dir1` and `dir2` — in the file system. To move the contents of `dir1` to `dir2`, a thread (e.g., `Thread_A`) needs to create two `Directory` objects — `objDir_1` and `objDir_2` — corresponding to `dir1` and `dir2`, respectively and pass them as arguments to the `moveContents` method.

```
//For Thread_A objDir_1 is the source directory
  moveContents(objDir_1, objDir_2);
```

While executing the `moveContents` method, `Thread_A` attempts to acquire locks on `objDir_1` and `objDir_2` in sequence.

At the same time, a different thread (e.g., `Thread_B`) invokes the `moveContents` method on the same `FileSysUtil` object to move `dir2` contents to `dir1`. Using the same `Directory` objects used by `Thread_A`, `Thread_B` makes a call as follows:

```
//For Thread_B objDir_2 is the source directory
  moveContents(objDir_2, objDir_1);
```

Similar to `Thread_A`, while executing the `moveContents` method, `Thread_B` also attempts to acquire locks on `objDir_1` and `objDir_2` but in the reverse order.

If `Thread_A` and `Thread_B` acquire locks at the same time on `objDir_1` and `objDir_2`, respectively, then each thread continues to wait for a lock on the `Directory` object locked by the other thread and this causes a deadlock. Because both `objDir_1` and `objDir_2` are of the same `Directory` class type, defining an object locking order based on the class type does not work in this case. As an alternative, the built-in Java `hashCode` method can be used to define an order of locking `Directory` objects. The `hashCode` method is defined in the topmost `java.lang.Object` class and is inherited by all classes in Java.

The hashCode method returns the unique ID or hash code associated with an object. An object locking scheme can be defined based on some kind of order of the hash codes of the objects to be locked.

To eliminate the possibility of a deadlock situation, the moveContents method can be modified so that the objects representing the source and the destination directories are locked in the ascending order of their associated hash codes. This ensures that the Directory objects are always locked in the same order, even if they are passed to the moveContents method by two different threads in different order.

```
        …
        …
  public void moveContents(Directory src, Directory dest) {
    if (src.hashCode() > dest.hashCode()) {
      synchronized (src) {
          synchronized (dest) {
            System.out.println("Contents Moved Successfully");
          }
        }
      } else {
         synchronized (dest) {
           synchronized (src) {
             System.out.println("Contents Moved Successfully");
           }
         }
       }
     }
        …
        …
```

With this change in place, when two threads invoke the moveContents method at the same time to move the contents of two different directories in opposite directions, only one thread is granted lock on the first Directory object to be locked. The second thread simply waits for the lock on the first Directory object itself. The possibility of the second thread locking the second Directory object while the first thread locks the first Directory object does not arise.

The example application uses a simple mechanism to define the locking order for Directory objects. In the case of a real world application, a locking order that is suitable for the application needs to be identified and documented. This locking order can then be followed consistently during the design and the development of the application.

PRACTICE QUESTIONS

1. Design a class `AccountManager` with a method to transfer money from one bank account to another. For this class to be used in a multithreaded environment, it must lock both the account objects before performing the actual transfer. Implement a method to transfer money so that when two different threads attempt to transfer money between two different accounts at the same time in opposite directions, it does not result in a deadlock in a multithreaded environment.
2. Design a class `InventoryManager` with a method to move products from one distribution center to another. For this class to be used in a multithreaded environment, it must lock the objects representing the two distribution centers that are participating in the transaction before performing actual updates to their inventory levels. The method to move products should be implemented in a manner that does not cause a deadlock when two different threads attempt to move items between two distribution centers at the same time in opposite directions.