Acquire multiple locks in a fixed, global order to avoid deadlock

Peter Haggar
Senior Software Engineer, IBM
September 2000

Editor's note: The following article is an excerpt from the book "Practical Java" published by Addison-Wesley. You can order this book from Borders.com. Read our interview with author Peter Haggar.

Deadlock occurs when two or more threads are blocked while waiting for each other. For example, the first thread is blocked on the second thread, waiting for a resource that the second thread holds. The second thread does not release this resource until it acquires a resource held by the first thread. Because the first thread cannot release its resource until it acquires one from the second thread, and the second thread cannot release its resource until it acquires one from the first thread, the threads are deadlocked.

Deadlock is one of the most difficult problems to handle in multithreaded code. Finding and fixing it is arduous and time consuming because it can occur in the least expected places. For example, consider the following code that locks multiple objects.

```
public int sumArrays(int[] a1, int[] a2)
{
  int value = 0;
  int size = a1.length;
  if (size == a2.length) {
    synchronized(a1) {                                    //1
      synchronized(a2) {                                  //2
        for (int i=0; i<size; i++)
          value += a1[i] + a2[i];
      }
    }
  }
  return value;
}
```

This code properly locks the two array objects before they are accessed in a summation operation. It is short, simple, and properly written for the task it performs, but unfortunately it potentially has a problem. The problem is that it creates a potential deadlock situation unless additional care is taken in how this method is invoked on the same objects from different threads. To see the potential deadlock, consider the following sequence of events:

1. Two array objects are created, `ArrayA` and `ArrayB`.
2. Thread 1 invokes the `sumArrays` method with the following invocation:
   `sumArrays(ArrayA, ArrayB);`
3. Thread 2 invokes the `sumArrays` method with the following invocation:
   `sumArrays(ArrayB, ArrayA);`
4. Thread 1 begins executing the `sumArrays` method and acquires the lock for parameter a1 at //1, which for this invocation is the lock for the `ArrayA` object.
5. Thread 1 is then preempted before acquiring the lock for `ArrayB` at //2.
6. Thread 2 begins executing the `sumArrays` method and acquires the lock for parameter a1 at //1, which for this invocation is the lock for the `ArrayB` object.
7. Thread 2 then attempts to acquire the lock for parameter a2 at //2, which is the lock for the `ArrayA` object. Thread 2

blocks because this lock is currently held by Thread 1.

8. Thread 1 begins executing and attempts to acquire the lock for parameter a2 at //2, which is the lock for the `ArrayB` object. Thread 1 blocks because this lock is currently held by Thread 2.

9. Both threads are now deadlocked.

One way to avoid this problem is for code to acquire locks in a fixed, global order. In this example, if thread 1 and thread 2 call the `sumArrays` method with the parameters in the same order, the deadlock will not occur. This technique, however, requires programmers of multithreaded code to be careful in how they invoke methods that lock objects passed as parameters. Application of such a technique might seem unreasonable until you encounter this type of deadlock and have to debug it.

Alternatively, you can have the lock ordering embedded within the object. This allows code to query the object it is about to acquire a lock for to determine the proper locking order. As long as all objects to be locked support the lock ordering notion and code that acquires locks adheres to this strategy, you avoid these potential deadlock situations.

The disadvantage of embedded lock ordering in objects is the extra memory and runtime costs associated with its implementation. In addition, applying this technique in the previous example requires a wrapper object on the arrays to contain the lock ordering information. For example, consider the previous modified code with an implementation of the lock ordering technique:

```
class ArrayWithLockOrder
{
  private static long num_locks = 0;
  private long lock_order;
  private int[] arr;

  public ArrayWithLockOrder(int[] a)
  {
    arr = a;
    synchronized(ArrayWithLockOrder.class) {
      num_locks++;              //Increment the number of locks.
      lock_order = num_locks;  //Set the unique lock_order for
    }                           //this object instance.
  }
  public long lockOrder()
  {
    return lock_order;
  }
  public int[] array()
  {
    return arr;
  }
}

class SomeClass implements Runnable
{
  public int sumArrays(ArrayWithLockOrder a1,
                       ArrayWithLockOrder a2)
  {
    int value = 0;
    ArrayWithLockOrder first = a1;       //Keep a local copy of array
    ArrayWithLockOrder last = a2;        //references.
    int size = a1.array().length;
    if (size == a2.array().length)
    {
      if (a1.lockOrder() > a2.lockOrder())  //Determine and set the
      {                                     //lock order of the
        first = a2;                         //objects.
        last = a1;
      }
      synchronized(first) {              //Lock the objects in correct order.
        synchronized(last) {
          int[] arr1 == a1.array();
          int[] arr2 == a2.array();
          for (int i=0; i<size; i++)
            value += arr1[i] + arr2[i];
        }
      }
    }
    return value;
```

```
  }
  public void run() {
    //...
  }
}
```

The `ArrayWithLockOrder` class is provided as a wrapper to the arrays used in the first example. This class increments the `static num_locks` variable each time a new object of the class is created. A separate `lock_order` instance variable is set to the current value of the `num_locks static` variable. This ensures that each object of this class has a unique value for the `lock_order` variable. The `lock_order` instance variable serves as the indicator for the order that this object should be locked in relation to other objects of this class.

Note that the manipulation of the `static num_locks` variable is done from within a `synchronized` statement. This is required because each instance of an object shares its `static` variables. Therefore, if two threads create an object of the `ArrayWithLockOrder` class concurrently, the `static num_locks` variable could be corrupted if the code manipulating it is not synchronized. Synchronizing this code ensures that each object of the `ArrayWithLockOrder` class has a unique value for its `lock_order` variable.

The `sumArrays` method is also updated to include code to determine the correct lock ordering. Before locks are requested, each object is queried for its lock order. Lower numbers are locked first. This code ensures that regardless of the order in which objects are passed to this method, they are always locked in the same order.

The `static num_locks` field and `lock_order` field are both implemented as a `long`. The `long` data type is implemented as a 64-bit signed two's complement integer. This means that the `num_locks` and `lock_order` values will roll over after 9,223,372,036,854,775,807 objects are created. It is unlikely that you will reach this limit, but it is possible under the right circumstances.

Implementing embedded lock ordering requires some extra work, memory, and execution time. However, you might find it worth the cost if these types of dead-lock situations are possible in your code. If you cannot afford the extra memory and execution overhead or the likelihood exists of rolling over the `num_locks` or `lock_order` fields, you should carefully establish a predefined order for locking objects.

About the author

Peter Haggar is a Senior Software Engineer with IBM. He currently works on emerging Java and Internet technology and is the project lead for IBM's real-time Java reference implementation. He has a broad range of programming experience having worked on development tools, class libraries, and operating systems. He is also a frequent technical speaker on Java and other technologies at numerous industry conferences. He received a B.S. in Computer Science from Clarkson University in New York in 1987. He can be reached at haggar@us.ibm.com.