The Art of Multiprocessor Programming
Solutions to Exercises
Chapter 18

July 14, 2009

Exercise 211. Implement the *Priority*, *Greedy*, and *Karma* contention managers.

Solution See code folder for this chapter.

Exercise 212. Describe the meaning of orElse without mentioning transaction roll-back.

Solution The transaction is very, very lucky and every time it is scheduled all shared objects just happen to satisfy the condition required by the orElse clause that happens to be executed.

Exercise 213. In TinyTM, implement the openRead() method of the FreeObject class. Notice that the order in which the Locator fields are read is important. Argue why your implementation provides a serializable read of an object.

Solution See Figure . The difference between openWrite() and openRead() is simple: openRead() does not need to create a new copy of the object.

Exercise 214. Invent a way to reduce the contention on the global version clock in TinyTM.

Solution Use any of the shared counter algorithms described in Chapter **??**.

Exercise 215. Extend the LockObject class to support concurrent readers.

Solution We keep track of readers in a thread-local ReadSet object (Figure 3). Figure 4 shows the code for openRead().

Exercise 216. In TinyTM, the LockObject class's onCommit() handler first checks whether the object is locked by another transaction, and then whether its stamp is less than or equal to the transaction's read stamp.

- Give an example showing why it is necessary to check whether the object is locked.
- Is it possible that the object could be locked by the committing transaction?
- Give an example showing why it is necessary to check whether the object is locked *before* checking the version number.

```
1  /**
2   * Interface satisfied by all contention managers
3   */
4  public interface ContentionManager {
5    void resolveConflict (Transaction me, Transaction other);
6    void resolveConflict (Transaction me, Collection<Transaction> other);
7    long getPriority ();
8    void setPriority (long value);
9    void openSucceeded();
10   void committed();
11 };
12 public abstract class BaseManager implements ContentionManager {
13   long priority ;
14   public BaseManager() {
15     priority = 0;
16   }
17   public abstract void resolveConflict (Transaction me, Transaction other);
18   public void resolveConflict (Transaction me, Collection<Transaction> other) {
19     for (Transaction t : other) {
20       if (me != t) {
21         resolveConflict (me, t);
22       }
23       if (me.isAborted()) {
24         break;
25       }
26     }
27   }
28   public long getPriority () {
29     return priority ;
30   }
31   public void setPriority (long value) {
32     priority = value;
33   }
34   public void openSucceeded() {
35   }
36   protected void sleep(int ns) {
37     long startTime = System.nanoTime();
38     long stopTime = 0;
39     do {
40       stopTime = System.nanoTime();
41     } while ((stopTime - startTime) < ns);
42   }
43   public void committed() {
44   }
45   public boolean isDeterministic () {
46     return false;
47   }
48 }
```

Figure 1: Contention Manager interface and base class used by other contention managers

```
1   public T openRead() {
2     Transaction me = Transaction.getLocal();
3     switch (me.getStatus()) {
4       case COMMITTED:
5         return openSequential();
6       case ABORTED:
7         throw new AbortedException();
8       case ACTIVE:
9         Locator locator = start.get();
10        if (locator.owner == me) {
11          return locator.newVersion;
12        }
13        Locator newLocator = new Locator();
14        while (!Thread.currentThread(). isInterrupted ()) {
15          Locator oldLocator = start.get();
16          Transaction writer = oldLocator.owner;
17          switch (writer.getStatus()) {
18            case COMMITTED:
19              newLocator.oldVersion = oldLocator.newVersion;
20              break;
21            case ABORTED:
22              newLocator.oldVersion = oldLocator.oldVersion;
23              break;
24            case ACTIVE:
25              ContentionManager.getLocal().resolve(me, writer);
26              continue;=\\lab
27          }
28          if (start.compareAndSet(oldLocator, newLocator)) {
29            return newLocator.newVersion;
30          }
31        }
32        me.abort(); // time's up
33        throw new AbortedException();
34      default:
35        throw new PanicException("Unexpected transaction state");
36    }ioe
37  }
```

Figure 2: The openRead() method of the FreeObject class

```
1  public class ReadSet implements Iterable <LockObject<?>> {
2    static ThreadLocal<Set<LockObject<?>>> local = new ThreadLocal<Set<LockObject<?>>>() {
3      protected Set<LockObject<?>> initialValue() {
4        return new HashSet<LockObject<?>>();
5      }
6    };
7    Set<LockObject<?>> set;
8    private ReadSet() {
9      set = local.get();
10   }
11   public static ReadSet getLocal() {
12     return new ReadSet();
13   }
14   public Iterator <LockObject<?>> iterator() {
15     return set.iterator();
16   }
17   public void add(LockObject<?> x) {
18     set.add(x);
19   }
20   public void clear() {
21     set.clear();
22   }
23 }
```

Figure 3: The ReadSet class used to solve Exercise 215

```
1    public T openRead() {
2      ReadSet readSet = ReadSet.getLocal();
3      switch (Transaction.getLocal().getStatus()) {
4        case COMMITTED:
5          return version;
6        case ACTIVE:
7          WriteSet writeSet = WriteSet.getLocal();
8          if (writeSet.get(this) == null) {
9            if (lock.isLocked()) {
10             throw new AbortedException();
11           }
12           readSet.add(this);
13           return version;
14         } else {
15           @SuppressWarnings(value = "unchecked")
16           T scratch = (T) writeSet.get(this);
17           return scratch;
18         }
19       case ABORTED:
20         throw new AbortedException();
21       default:
22         throw new PanicException("unexpected transaction state");
23     }
24   }
```

Figure 4: The LockObject class: supporting concurrent readers

Solution Here is why a transaction must check that each object in its read set is unlocked. There are two transactions, A and B, and an object x. Initially, x has stamp 0. A observes global version clock 1, updates x, then, on commit, locks x, increments the global version counter, but pauses before updating x's stamp. B observes global version clock 2, and reads x. If B does not check whether x is locked, it will validate version 0 of x. A then commits, installing a new version of x with stamp 1. B must be serialized before A, because it read an older version of x, but it has a higher global version clock value.

Yes, it is possible that x is locked by the validating transaction, which does not indicate a conflict.

If B checks the object's tamp before its lock, then in the scenario just described, B observes x to have stamp 0, A locks x, advances the stamp, and unlocks. B then observes that x is unlocked, and validates incorrectly.

Exercise 217. Design an AtomicArray<T> implementation optimized for small arrays such as used in a skip list.

Solution We could adapt either the LockObject or FreeObject classes to hold old and new versions of the entire array. Because they array is small, it is acceptable to copy the entire array.

Exercise 218. Design an AtomicArray<T> implementation optimized for large arrays in which transactions access disjoint regions within the array.

Solution If the array is large, then it is inefficient to copy the entire array on each update. Instead, it is more efficient to treat the array as a sequence of smaller blocks, where synchronization and recovery are performed at the block level.