Lecture 06
Object-Oriented Concurrent Programming

Shared objects

Object-Oriented Concurrent Programming, 2019-2020

v2.3, 24-10-2019

Miguel Oliveira e Silva, DETI, Universidade de Aveiro

Contents

1 Immutability

- If an object is completely immutable (i.e. immutable, and transitively immutable), then all correctness conditions apply when it is shared;

Immutable Shared Objects: Immutable objects are *thread-safe*!

- In Java, immutable shared objects are thread-safe iff:
  - its state cannot be changed after construction;
  - all its fields, if any, are `final`;
  - if it is build in a way that ensures that its referece does not escapes to the outside inside constructors.
- A consequence of immutability is the fact that assertions retain their sequential semantics (by definition, because no other thread can change its value).

Immutable Objects: example

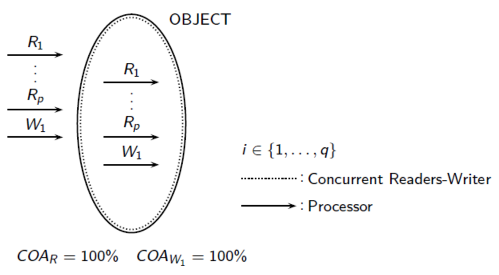
```java
@Immutable
public class Point {
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }

   ...

   private final int x,y;
}
```

```java
@Immutable
public class Line {
   public Line(Point p1, Point p2) {
      assert p1 != null && p2 != null;
      assert !p1.equals(p2);
      this.p1 = p1;
      this.p2 = p2;
   }

   ...

   private final Point p1,p2;
}
```

2 Internal synchronization schemes

2.1 Concurrent Readers-Writer

- Take advantage of a single “writer” thread (when exists).
- That is, the shared object commands are only executed by the same unique thread throughout its lifetime.



2.1.1 Implementation

- This synchronization scheme can be implemented by attaching two atomic counters to commands (that, remember, can only be executed by the same thread throughout the object’s lifetime). One counter counts the number of command entries and the other the number of exits.
- The implementation of queries is slightly more complicated. First the possibility of a query failure, as a result of a simultaneous execution of a command, must be considered. The solution is to (internally) repeat the execution of the query until it succeeds. A query is successful if, and only if, both counters have the same value, and this value equals the exit counter value stored in the beginning of the query.
- This scheme is restricted to the limitation that there is only a single writer thread during the object’s lifetime (hence it is not as general as previously presented schemes).

- In the implementation of this scheme, it must be considered the possibility of a query failure due to a concurrent execution of a command (i.e. a race condition) might result in throwing an exception.
- Hence, queries must be invoked inside a `try-catch` block, catching all exceptions. Whenever an exception is caught it should only be propagated if the counters success condition is met. Or else, the query should be transparently repeated (as explained previously).

Concurrent Readers-Writer - Contracts

- A simple analysis of this synchronization scheme shows that assertions will be sequential for the writer (because he is the only one that could concurrently change its value); and are (possibly) concurrent for readers.

2.1.2 Example

Note that the following code only addresses the problem of internal object synchronization. A full implementation must also consider the external and conditional synchronization. Nevertheless, conditional synchronization raises a curious problem. With only a writer thread, a command will never require conditional synchronization (at least, for internal reasons). That is, in commands assertions involving the object's state are always sequential.. On the other hand, queries might require a conditional synchronization.

```java
import java.util.concurrent.atomic.AtomicInteger;
public class RWCStack<T>
   extends SharedStack<T> {
   ...
   public void push(T e) {
      countIn.incrementAndGet();
      stack.push(e);
      countOut.incrementAndGet();
   }

   public T pop() {
      countIn.incrementAndGet();
      T result = stack.pop();
      countOut.incrementAndGet();
      return result;
   }

   protected AtomicInteger
      int countIn = 0;
   protected AtomicInteger
      int countOut = 0;
   ...
```

```java
   public int size() {
      int result = false;
      boolean repeat = false;
      int count = 0;
      do {
         try {
            count = countOut.get();
            result = stack.size();
            repeat = (count != countIn.get());
         }
         catch(Throwable e) {
            if (count != countIn.get())
               repeat = true;
            else
               throw e;
         }
      }
      while(repeat);
      return result;
   }
}
```

2.2 *Copy-on-Write*

- The concurrent readers-writer synchronization scheme is interesting because is relatively simple and allows total concurrent availability of the shared object. However, it is limited to a single writer thread in all the object's lifetime (which is a very strong restriction).
- An interesting alternative that retains total concurrent availability, but without restricting the number of writer threads, are *copy-on-write* synchronization schemes (CoW).
- These schemes are based on the simple idea of considering each object as a temporal succession of immutable stable states (with atomic transitions).
- Implementations based on this principle, ensure that all threads observe shared objects in stable times, and might not require any other synchronization beyond the one imposed by the language memory model.

- To implement this behavior commands will have to work on a copy of the object's state, and making it atomically the new object's state when the command ends.
- There are two possible approaches in the atomic update of the object's state:

  1. "pessimistic": writer threads work in exclusion with each other (but not with reader threads) which ensures the success of the state update;
  2. "optimistic": writer threads work concurrently, a command might fail if another command has submitted a new object's state. In this case, it is necessary to repeat the execution of the command until it succeeds.

- The first case corresponds to the presented CoW scheme.
- In the second case, we are in the presence of an internal software transaction synchronization scheme.
- Note that, unlike readers-writer exclusion, in a CoW scheme readers are never blocked from using the shared object.

CoW algorithm (case 1):

```java
public class CoWClass {
   public synchronized void command(...) {
      ObjectType clone = obj.deepClone(); // atomic copy!!!
      clone.command(...);
      obj = clone; // atomic copy
   }

   public Type query(...) { // Note that is not synchronized!!!
      Type result;
      ObjectType current = obj; // atomic copy!!!
      // obj query operations
      return result;
   }

   ...

   protected volatile ObjectType obj;
}
```

- Note the of of `synchronized` in method `command` to ensure exclusion between writer threads (pessimistic approach).
- Also, Java memory model ensures that it is enough to use a volatile object variable to achieve the desired behaviour.

- This generic algorithm can be simplified if we restrict the internal state of objects to a succession of *immutable states*.
- In this way, we can avoid the duplication of states (which might be a heavy operation).

```java
public class PointCOW {
   public synchronized void move(int dx, int dy) {
       point = new ImmutablePoint(point.x()+dx, point.y()+dy);
   }

   public int x() {
      return point.x();
   }

   protected volatile ImmutablePoint point;
}
```

- A very interesting particular application of this scheme is the implementation of a stack with a linear single linked list.
- In this kind of list the inclusion of a new element (`push`) consists in the atomic append of a new node in which the next reference is the previous list.
- If nodes are immutable, there is no need to copy the entire list, and this operation is very efficient.
- On the other hand, the removal of the top node (`pop`), consists only in updating the top reference of the stack to the next list (assuming, of course, that the list is not empty).
- These operations have $O(1)$ algorithm complexity, and don't require an integral copy of the list.

```java
public class ImmutableLinkedList<T> {
   public ImmutableLinkedList(T head, ImmutableLinkedList<T> tail) {
      this.head = head;
      this.tail = tail;
   }

   public final T head;
   public final ImmutableLinkedList<T> tail;
}
```

```java
public class LinkedListCoW<T> {
   public synchronized void prepend(T e) {
      list = new ImmutableLinkedList<T>(e, list);
   }

   public ImmutableLinkedList<T> current() {
      return list;
   }

   protected volatile ImmutableLinkedList<T> list=null;
}
```

2.3 Transactions

The second approach presented – optimistic updating of the object internal state – is a scheme of software transactions.

- An optimistic approach to the CoW problem allows the possibility of not blocking writer threads, but it requires the possibility of (internally) repeating command execution whenever the transaction fails.
- This kind of schemes requires that there cannot be any external collateral effects on the execution of commands (such as, writing to the console or to files), and may raise problems of systematic interference between threads (*livelocks*), although it ensures that at least one transaction always succeeds.
- To alleviate the problem of systematic thread interference, a random growing delay strategy can be implemented (similar to the *back-off* exponential delay used in *ethernet* network buses).
- Lock-free synchronization schemes such as transactions have the advantage of preventing that a threads blocks the access to a shared object, and the disadvantage of a possible lower efficiency (due to the use of extensive copying) and livelocks.

Transactions: Example

```java
public class IntraTransactionClass {
   public void command(...) { // Note that is not synchronized!
      boolean retry;
      do {
         ObjectType old = obj; // atomic copy!
         ObjectType clone = old.deepClone();
         clone.command(...); // External side-effects not allowed!
         synchronized(this) { // Atomic Compare-and-Swap!
            retry = (obj != old);
            if (!retry)
               obj = clone; // atomic copy
         }
      } while(retry);
   }

   public Type query(...) { // Note that is not synchronized!
      Type result;
      ObjectType current = obj; // atomic copy!!!
      // obj query operations
      return result;
   }

   ...

   protected volatile ObjectType obj;
}
```

2.4 Internal and Conditional Synchronization

- So far we have showed that a safe internal synchronization can be achieved with very different synchronization schemes, from the simpler more restrict mutual exclusion up to the more complex but much more permissive lock-free schemes.
- Also, we have showed how to safely allow any internal scheme with an exclusion external synchronization scheme (group mutual exclusion).
- What about (internal) conditional synchronization?
- We already know that the need to use this synchronization arises from concurrent contracts.
- Can conditional synchronization be also lock-free, or blocking is unavoidable?

- The first thing we must realize, is that the necessity for conditional synchronization is part of the programs algorithmic logic, and not from some timing constraints related with shared resources or thread scheduling.
- Hence, it makes little sense to allow lock-free to a condition that can only be verified by the logical execution of the program's code.
- Thus, when required by the program semantics (concurrent assertions), conditional synchronization requires an algorithmic blocking behavior.
- If this behavior is not, for same reason, desirable. Then one must weaken the algorithms contract (for instance, delegating previous precondition "failure" handling to methods clients), or add some finite maximum time blocking, and handle failure with appropriate error tolerance algorithm.

2.5 Summary of internal object synchronization schemes

	command				**query**				**mode**
Scheme	best	worst	command	query	best	worst	command	query	
Monitor	100%	0%	0%	0%	100%	0%	0%	0%	block
RWEx	100%	0%	0%	0%	100%	0%	0%	100%	block
RWC	100%	0%	0%	100%	100%	100%	100%	100%	repeat
CoW	100%	0%	0%	100%	100%	100%	100%	100%	block
transaction	100%	100%	100%	100%	100%	100%	100%	100%	repeat

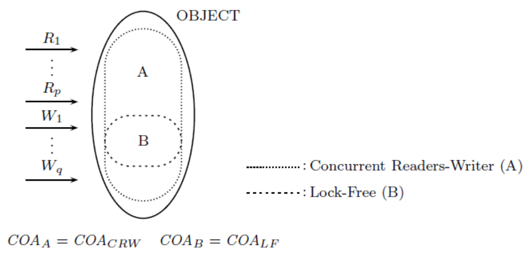
- This table shows the concurrent availability of different synchronization schemes.

- The `command` and `query` columns show the concurrent availability when the operation is running concurrently with a command or query (respectively).
- The last column (`mode`) shows if the synchronization scheme is pessimistic (i.e. blocks until it ensures that the synchronization operation succeeds), or optimistic (i.e. repeats on failure).
- Note that repetition is only possible if there is no external side effects on the execution of the operation.

3 Mixed Synchronization Schemes in Exclusion

- As an informal general rule, when the concurrent availability of shared objects increases, so does the complexity of the synchronization scheme (either in its code or in its algorithm complexity), and also increase the restrictions posed to the source code.
- On the other hand, regarding the correctness criteria applicable to shared objects, there is no theoretical reason (or practical) to force the usage of a single synchronization scheme on a shared object.
- Under certain conditions, we may allow the use – in exclusion – of different synchronization schemes.
- In fact, we use this approach to find a solution to the coexistence of different internal and external synchronization schemes.
- In that case, switching between synchronization schemes was temporal.
- However, we can use the same strategy – group exclusion – to implement mixed synchronization schemes, in which several schemes coexist within the object (either in temporal exclusion, or applicable to a subset of the object state).

- The correctness criteria to the use of mixed synchronization schemes are the following:
  1. The mixed scheme must ensure total object covering;
  2. All the schemes work in exclusion of each other;
  3. Each synchronization sub-scheme must be safe in the whole range of services to which it applies;
  4. Each subset of services synchronized by a sub-scheme cannot depend (invoke) on services that are not synchronized by the same sub-scheme.


OBJECT
R_1
R_p
W_1
W_q
A
B
: Concurrent Readers-Writer (A)
: Lock-Free (B)
$COA_A = COA_{CRW}$
$COA_B = COA_{LF}$