Quiescent Consistency

Michal Misiak

2015-05-03

Abstract

An execution of a concurrent program is *quiescently consistent* if the method calls can be correctly arranged retaining the mutual order of calls separated by quiescence, a period of time where no method is being called in any thread.

Quiescent consistency semantics impose less restrictions than strong consistency model — *linearizability*, allowing for greater freedom in implementing concurrent programs. It is appropriate for applications that require high performance at the cost of placing relatively weak constraints on object behaviour.

The implications of the relaxed constraints and various aspects of quiescent consistency are described in this article.

All the examples are written in Java 1.8 and tested using Java HotSpot(TM) virtual machine.

1 Concurrent Execution

*Quiescent consistency* is a correctness model of concurrent programs (another important property is its liveness but it is beyond the scope of this article). There are other correctness models, e.g. *sequential consistency* or *linearizability*.

One way of expressing correctness of an execution of a concurrent program is by referring to the correctness of its sequential (i.e. single-threaded) execution. Assuming that correct sequential executions are known (defined by some means — a specification), the correctness of a concurrent execution may be defined by mapping (reordering) it to a sequential execution. Different mapping rules (valid reorderings) define different semantics, e.g. quiescent consistency, linearizability, sequential consistency.

A sequential execution can be thought of as a sequence of method calls with the arguments and returned values, e.g. \( q \text{.add}(5) \to (); \) \( q \text{.get()} \to 5 \) (\( \to \) denotes returned value, \( () \) denotes void). Intuitively this execution seems correct for a queue \( q \) and the one \( q \text{.remove()} \to 5; \) \( q \text{.add}(5) \to () \) is not (assuming that the \( q \) is empty at the beginning). A sequential execution can be depicted like below.

\[
- [ \text{add}(5) \to () ] - [ \text{remove()} \to 5 ] -
\]

A concurrent execution is slightly more complicated since method invocations and returns may be interleaved with another thread invocations or returns. It can be formalised as a sequence of method invocations (with arguments) and returns (with returned values) together with a thread. So a sample concurrent execution of the queue methods can look like \( A: \) call \( q \text{.add} 5; B: \) call \( q \text{.remove}(); A: \) return \( q \text{.add} () ; B: \) return \( q \text{.remove} 5, \) where \( A \) and \( B \) are thread names and \( q \) is an object instance (queue). It can be depicted like below:

\[
A: - [ \text{add}(5) ] --------
B: ---- [ \text{remove()} \to 5 ] -
\]

Such an execution (also called a *history*) is considered correct if there exists a mapping satisfying a set of rules (those rules will be defined for quiescent consistency in the next section; they depend on the semantics, there are different rules for linearizability and sequential consistency) into a sequential execution that is correct.
For more details about various correctness semantics see "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit.

2 Quiescently Consistent Semantics

The key to understand quiescent consistency is a concept of quiescence. Informally a quiescence is a period when no method call takes place (on a particular object instance or set of instances). Formally a quiescence is a point in an execution where all method invocations can be paired with their corresponding method returns, i.e. there are no pending method calls. For example in execution:

1) A: call q.add(5)
2) B: call q.add(3)
3) B: return q.add()
4) A: return q.add()
5) A: call q.remove()
6) A: return q.remove 5

a quiescence is between 4 and 5 and after 6.

Please notice that the definition of quiescence may involve more than one object. See also section 4.

An execution is quiescently consistent if there exists a (correct) sequential execution that can be obtained by reordering the concurrent execution in between quiescence periods.

Whether a sequential execution is correct depends on a specification. The same sequential execution may by correct for a FIFO queue specification but incorrect for a FILO queue. It is assumed that correct sequential executions are known (given by some form of a specification). Knowing correct sequential executions, a quiescently consistent concurrent executions are those which can be mapped (reordered) into a correct sequential execution.

A quiescently consistent reordering of the execution above could be any order of steps 1 to 4 (that is sequential) followed by 5 and 6, e.g.

1) A: call q.add(5)
2) B: call q.add(3)
3) B: return q.add()
4) A: return q.add()
5) A: call q.remove()
6) A: return q.remove 5

is a permitted reordering (notice the difference between a permitted reordering and correct sequential execution). Another allowed reordering would be:

2) B: call q.add(3)
3) B: return q.add()
1) A: call q.add(5)
4) A: return q.add()
5) A: call q.remove()
6) A: return q.remove 5

is another valid reordering. The former is a correct sequential execution of a FIFO queue, the latter of a FILO queue.

Quiescently consistent semantics permits any reorderings of methods calls in between quiescence periods. Consider a single thread A executing some methods (on an object q). Since there is only one thread, there is a quiescence period between each method return and the following invocation. Those quiescence periods ensure that the method calls cannot be reordered. Consider another thread B executing one method (on the object q) that spans the lifetime of all thread A calls. This removes all the quiescence periods (except from the last one) allowing for any reordering. For example, let thread A be calling remove and add methods on a queue (the exact arguments and returned values are not important therefore they are omitted):

A: - [ remove ] --- [ add ] -

Such an execution is an illegal sequential execution for a queue, an element must first be added and only then removed (assume it is a FIFO queue, for producers and consumers). There is a quiescence period between the two methods (so they cannot be reordered) therefore this execution is not quiescently consistent either. But if there is another thread B calling e.g. method size which spans the lifetime of the methods remove and add:

A: - [ remove ] --- [ add ] -
B: ---- [ size ] ----

it becomes a legal (for a FIFO queue), quiescently consistent execution as there is no quiescence period between
remove and add and they can be reordered into a correct sequential execution.

See section [3] for an example of a program which exposes this "weird" behaviour.

See http://coldattic.info/shvedsky/pro/blogs/a-foo-walks-into-a-bar/posts/88 and "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit for other consistency models.

3 Order Matching

Let us consider a simple order matching class — Order. It has two main methods: offer and accept. The former is invoked by a producer and the latter by a consumer. Both methods take the number of items as an argument and return true if it was successful and false if the call was interrupted by calling the stop method. The offer method returns only if there are enough consumers willing to accept the items or enough free space in the buffer — capacity. This is to avoid production if there are no consumers or not enough space in the buffer. The accept method returns only if there are producers ready to offer the required number of items.

A simple implementation of the class may look like below. Please notice it may not be the most efficient implementation, its main purpose is to illustrate quiescently consistent semantics.

While analysing the code please ignore conc variable. It is measuring the number of attempts of the compareAndSet (contention) calls and it will be used in section [6].

```java
public class Order {
    private final int capacity;
    private final AtomicInteger items = new AtomicInteger();
    private volatile boolean stop = false;
    public AtomicLong conc = new AtomicLong();

    public Order(int capacity) {
        this.capacity = capacity;
    }

    public boolean offer(int num) {
        for (; ; ) {
            int i;
            do {
                if (stop)
                    return false;
                i = items.get();
                conc.incrementAndGet();
            } while (!items.compareAndSet(i, i + num));
            if (stop) {
                do {
                    i = items.get();
                } while (!items.compareAndSet(i, i - num));
                return false;
            }
            if (items.get() <= capacity)
                return true;
            // rollback
            do {
                i = items.get();
                conc.incrementAndGet();
            } while (!items.compareAndSet(i, i - num));
        }
    }

    public boolean accept(int num) {
        int i;
        do {
            if (stop)
                return false;
            i = items.get();
            conc.incrementAndGet();
        } while (!items.compareAndSet(i, i - num));
        if (items.get() <= capacity)
            return true;
        // rollback
        do {
            i = items.get();
            conc.incrementAndGet();
        } while (!items.compareAndSet(i, i - num));
    }
}
```
while (i < num ||
   !items.compareAndSet(
       i, i - num));
return true;
}

public void stop() {
    stop = true;
}

public void reset() {
    items.set(0);
    stop = false;
}
}

The offer method starts with increasing the number of items available, and in the beginning it disregards the capacity. It is designed that way in order to notify the waiting threads on the accept method as soon as possible so they can start consumption (and decrease items). Only then there is a check if the capacity have not been exceeded. If not the method returns with success. If it was exceeded the rollback process starts before another attempt. The rollback is necessary to avoid deadlocks between two producers.

The accept method simply waits until there are enough items to consume and if so it decreases the number of available items and returns with success.

The methods are not symmetric. The offer methods tries to signal the readiness of producers even if the capacity is exceeded. The accept method however does not goes below 0 to inform producers about the consumers waiting.

Both methods constantly check if the stop flag was set and if so, they return immediately with failure.

Additionally there is the reset method which resets the state of the monitor. It is added only for testing.

Let us test the implementation. Let thread A be accepting and then offering, and thread B be only offering. A simple code for A could be (let o be an instance of the Order class):

```java
o.accept(1);
o.offer(1);
```

and the one for B:

```java
o.offer(1);
```

The test will be run against Order instance with buffer of capacity 1.

The test code is a bit more complicated than the one presented above since it contains all the logic to invoke the scenario multiple times and to synchronise the start and the end of the test, reset the monitor state, etc. It also tries to detect which of the two threads completed first.

```java
public class OrderApp {
    public static void main(String[] args) throws InterruptedException {
        Order o = new Order(1);
        AtomicBoolean stop = new AtomicBoolean();
        AtomicInteger start = new AtomicInteger();
        AtomicInteger finished = new AtomicInteger();
        AtomicInteger end = new AtomicInteger();
        AtomicBoolean quiescence = new AtomicBoolean();

        Thread A = new Thread(() -> {
            for (; ; ) {
                start.incrementAndGet();
                while (!stop.get() &&
                    start.get() != 3);
                if (stop.get())
                    return;
                o.accept(1);
                o.offer(1);
                o.stop();
                finished.incrementAndGet();
                while (finished.get() != 3);
                end.incrementAndGet();
                while (end.get() != 3);
            }
        });
```
Thread B = new Thread(() -> {
    for (; ;) {
        start.incrementAndGet();
        while (!stop.get() &&
            start.get() != 3);
        if (stop.get())
            return;
        if (!o.offer(1)) {
            quiescence.incrementAndGet();
        }
        o.stop();
        finished.incrementAndGet();
        while (finished.get() != 3);
        end.incrementAndGet();
        while (end.get() != 3);
    }
});

A.start();
B.start();
for (int i = 0; i < 1000000; ++i) {
    start.incrementAndGet();
    while (start.get() != 3);
    end.set(0);
    finished.incrementAndGet();
    while (finished.get() != 3);
    start.set(0);
    o.reset();
    end.incrementAndGet();
    while (end.get() != 3);
    finished.set(0);
}
stop.set(true);
A.join();
B.join();

System.out.println(quiescence.get());
System.out.println(
o.conc.get() / 1_000_000d);
}

The synchronisation logic seems a bit complex but it assures that both threads start each cycle at exactly the same time. The variable start is a kind of semaphore on which the threads (A, B and the managing thread) spin to start the test. The stop flag set to true indicates the end of the test (all cycles).

Once started the threads (A and B) execute the Order class methods. Thread B which invokes only the offer method also counts the number of interruptions (failures) on the quiescence variable.

After the execution of the accept and offer methods, the threads synchronise on the finished mutex. This allows the managing thread to reset the state of the Order object. Once done, all the threads again synchronise on the end variable before they start a new cycle.

At the end of the test the number of interruptions (quiescence) in thread B is written to the standard output. Surprisingly, in most cases, it is not 0.

Thread B can be only interrupted by thread A which calls the stop method after accept and offer methods. If thread A completes first (before thread B), it means that somehow it managed to match its accept to its own offer from the future. An execution that may never happen if thread A was running alone (without B). The mere presence of thread B executing the offer method that spans the lifetime of thread A execution of its methods changed the behaviour in a "quiescently manner".

Indeed, quiescently consistent semantics allows any reordering of method executions between quiescent periods (i.e. periods when no methods are executed). Such reordering is not allowed in linearizable execution and thus the Order class is not linearizable but it is quiescently consistent.

A: --- [ accept ] - [ offer ] ---
B: --------[ offer ] --------

The presence of the offer method running in thread B, overlapping with methods accept and offer in thread A
allows reordering of the methods adhering to quiescently consistent semantics: A:offer, A:accept, B:offer.

A: --- [ offer ] - [ accept ] -----------
B: ---------------------------------- [ offer ... 

4 Compositional Property

Let us consider an execution E. By definition it is a sequence of method invocations and returns. It may involve one or many objects. Let a be an object. A subexecution of E on a is a subsequence of E that is targeted at a (method invocations on object a or returns from methods on a). For example, for an execution:

B: call q.add(3)
A: call p.remove()
A: return p.remove 5
A: call p.add(5)
A: return p.add ()
B: return q.add ()

its subexecution on p is:

A: call p.remove()
A: return p.remove 5
A: call p.add(5)
A: return p.add ()

A correctness property is compositional if, from the fact that each subexecution on each object involved is correct it follows that the execution is correct.

It is an important property that allows to verify program correctness "locally", for each object (class) independently and to build complex systems from small components.

Quiescently consistent semantics is compositional.

To prove that fact, it is enough to observe that an execution E has less or equal quiescence periods than any of its subexecutions (on any object). It means that any valid reordering on any subexecution is also valid reordering on E. In other words, the more happens, the more freedom is allowed.

In the example above, the execution is quiescently consistent as there is a call on q that spans the entire history (execution) and allows to reorder remove and add methods invoked on object p. But the subexecution on p is not quiescently consistent as there is a quiescence period between remove and add.

This shows that the opposite fact does not hold: if an execution is quiescently consistent it does not necessarily mean that each of its subexecutions is quiescently consistent.

5 Observability

Correctness of an object implementation is examined through its behaviour; a queue implementation is correct as long as its add and remove methods behave as expected. It disregards object’s internal state, only its exposed, observable behaviour matters.

Such an approach has some implications worth discussing.

Let us reconsider the Order class. There is a public attribute conc. It was made public to avoid adding a getter method. Let us add it:

```java
public long getConc() {
    return conc.get();
}
```

Let thread A be calling accept method and let thread B be calling offer method (concurrently). After that let some other thread C be calling getConc method. Let the buffer be again of size 1.

A concurrent execution may look like below:

A: - [ accept ] ------------
B: ---- [ offer ] -----------
C: ------------------ [ getConc ] -

What is the value returned by the getConc method? It depends. It may be 2 if the threads did not interrupt each others with compareAndSet. But it may be also any positive number. Assume it is 5. It is perfectly viable option for a concurrent execution. But is it quiescently consistent? If so there must be a sequential execution of the methods that is a reordered sequence of the concurrent calls that preserves the quiescence periods. It is easy to see that the only valid sequence is the below:
1) offer(1) -> true
2) accept(1) -> true
3) getConc() -> 5

But there is no sequential execution in which getConc methods returns 5. It will be always 2 as compareAndSet will always succeed in a single-threaded execution.

Despite what was said above it is still a correct quiescently consistent implementation. The fact that there exist no sequential execution of the methods as implemented in class Order that returns 5 does not make it incorrect since a correct sequential execution is an abstract thing. Correct sequential executions is a set of sequences of method calls that one considers correct. It is a specification. Simply the set of correct executions contains all sequences of the three methods (offer, accept and getConc) with all possible return values of the method getConc, e.g.:

1) offer(1) -> true
2) accept(1) -> true
3) getConc() -> 5

but also:

1) offer(1) -> true
2) accept(1) -> true
3) getConc() -> 5

and many others.

Correct executions are abstract beings. They are defined by some form of a specification (formal or informal), and may but do not have to be given by a working implementation (specification by example).

Similar example exists in the "sequential world". Think of a quick sort implementation with a random pivot. The implementation may measure how good the choices of the split elements were by calculating the average ratio of the split. If the class implementing the algorithm exposes the ratio via a method (a getter) it also must be allowed to return any (reasonable, i.e. between 0 and 1) result.

6 Limitations

This section does not address the limitations of the quiescence consistency but generally it refers to any concurrent specifications that express correctness of concurrent programs by mapping it to a sequential execution.

Each quiescently consistent executions are atomic, i.e. method calls appear to take effect instantaneously.

Improperly implemented queue may for example, when two threads execute add method concurrently, one with -1 argument and the other with 5, enqueue -5 (i.e. subsequent remove call may return -5). Such executions are excluded by the simple fact that the execution (a sequence) must be mapped to a sequential execution. It does not depend on the particular reordering rules, it applies to both quiescently consistent semantics and linearizability.

Some correct implementation may not be expressed using quiescently consistent semantics (or any other semantics that refers to a sequential execution).

Consider again the Order class instance with capacity 0. When there is no buffer the class actually matches producers to consumers, i.e. the accept method will finish only if there is another thread calling the offer method. It simply pairs consumer threads with producer threads. And it may be useful sometimes. But it does not make sense to think of it sequentially as the sequential execution will always block.

It is obvious that such behaviour, concurrent in nature (a method finishes only when there is another thread executing another method), may not be expressed by referring to a sequential executions. Quiescent consistency does not also guarantee any liveness property; if there are many threads simultaneously calling offer and accept methods they may theoretically spin forever. This is however beyond the scope of this article.