

1 Concurrent Objects

In this lecture we will discuss the correctness (safety) properties of concurrent data objects. The key idea, as given in [1], is that it is easier to reason about these objects by mapping their concurrent executions to sequential ones, and limiting the reasoning to these sequential executions. So the correctness properties will be described as constraints on the sequential ordering of concurrent executions. The three properties we will cover are:

1. Quiescent Consistency (*QC*)
2. Sequential Consistency (*SC*)
3. Linearizability (*LIN*)

While *QC* and *SC* are not comparable, i.e. $QC \not\Rightarrow SC$ and $QC \not\Leftarrow SC$, *LIN* is stronger than both of them.

1.1 Sequential specification of data objects

We can describe the desired behavior of an object—in absence of concurrency—by specifying what kind of sequences of operations¹ are legal for that object. This is the *sequential specification* of the object and it consists of a set of operations defined on the object, and a set of sequences of operations. It is based on the effect on the object's state by the various operations.

For example, for the FIFO queue:

state : a sequence of elements, possibly empty.

ENQ(*z*): changes state σ to $\sigma.z$

DEQ() : changes state $a.\sigma$ to σ and returns *a*. Requires that queue be non-empty.

Using above descriptions of *ENQ* and *DEQ* we can specify the following two rules that determine what sequences of operations are legal for a FIFO queue:

1. For each *DEQ*() that returns *x*, there must be a corresponding *ENQ*(*x*) preceding it.
2. The subsequence of *DEQ*() operations must be a prefix of the subsequence of the *ENQ*() operations.

¹Operations are described in more detail in the next section.

Another example: RW registers. Here the operations are *READ* and *WRITE*. The sequential specification says that a sequence of operations is legal if each *READ* returns the value of the most recent preceding *WRITE*.

1.2 Operations on objects

Operations are pairs of invocations and matching responses. An operation begins at the invocation and ends at the matching response. For example, the two operations of a RW register called X have invocations and response pairs as follows:

operation	invocation	response
$\text{READ}_i(X, v)$	$\text{read}_i(X)$	$\text{return}_i(X, v)$
$\text{WRITE}_i(X, v)$	$\text{write}_i(X, v)$	$\text{ack}_i(X)$

Definition 1 *Correct interaction:* For each processor P_i , and some sequence of events σ , $\sigma|i$ consists of alternating invocations and matching responses beginning with an invocation.

Definition 2 *A quiescent state for a object is when there is no operation pending on that object.*

1.3 The three correctness properties

With the above discussion of background information, we now consider the three correctness properties for objects, in turn, in more detail.

1.3.1 Quiescent consistency (QC)

This property is defined by the following two conditions:

1. Operations should appear in some sequential order (legal for each object).
2. Operations whose occurrence is separated by a quiescent state should appear in the order of their occurrence.

QC provides high-performance at the expense of weaker constraints provided by the system.

1.3.2 Sequential consistency (SC)

This property is satisfied if there exists a permutation π of the operations in σ such that

1. For every object X , $\pi|X$ is legal, according to the sequential specification of X .
2. If the response for operation o_1 at node P_i occurs in σ before the invocation for operation o_2 at node P_i , then o_1 appears before o_2 in π .

This property is useful for describing low-level systems such as hardware interfaces.

Sequential consistency is only concerned with preserving local order of operations for each processor. It does not respect the global ordering of the operations across processors. The example below demonstrate a violation of global order by a sequentially consistent execution. Here A and B are processors (program threads) and X is a read/write register object.

```
A -----[ X.write(1) ]---[ X.write(2)]-----  
B -----[ X.read(1) ]--
```

This execution is sequentially consistent, since we can reorder the operations as follows:

{X.write(1) at A}, {X.read(1) at B}, {X.write(2) at A}

This reordering respects the program orders of A and B individually, but does not respect global ordering of the operations, since in the actual execution, operation {X.write(2) at A} finishes before {X.read(1) at B} begins.

1.3.3 Linearizability (*LIN*)

This property is satisfied if there exists a permutation π of all the operations in σ such that

1. —Same as condition 1 of *SC*—
2. If response of operation o_1 occurs in σ before invocation of operation o_2 , then o_1 appears before o_2 in π .

This property is useful in describing higher-level systems composed from linearizable components.

Note that condition 2 of *LIN* is stronger than *SC*'s condition 2; it implies that linearizability respects the global order of operations.

References

- [1] Maurice Herlihy, and Nir Shavit, *The Art of Multiprocessor Programming*, Morgan and Kaufmann, Burlington, MA 2008.