Linearizability is a correctness criterion for a program containing concurrent accesses to shared objects. It is in a sense a concise way of saying a program is correct.

Linearizability assumes that each operation takes effect at some instant between the the moment the operation is invoked and the moment of the response.

Let’s examine the following example. There are three processes (A,B,C) executing read/write operations on a single shared register (x). The limits of the intervals in the following diagram represent the invocation/response moments of each operation, as seen by an external observer. We want to see what value will be read by the last read operation performed by process B.

For the diagram above process B can only read a value of 0 (since A read the 1 value written by C, and its response came before the invocation of B’s write operation). The only possible linearization is: A’s write, C’s write, A’s read, B’s write. On the other hand, if A’s read operation would extend further to the right, B might read a value of 1. A possible ordering would be the following: A’s write, B’s write, C’s write, A’s read. As long as the value read in the end can be observed after a linearization of the sequence of the operations seen before, the program is considered correct.

The notion of linearizability can be formalized in terms of histories. A history is a series of method invocations and responses. For instance, the diagram above corresponds to the following history:
The OKs represent method responses (their “parameter” represents the result generated by the corresponding invocation). A method invocation together with its corresponding response form an operation. In the case of interleaving operations, an invocation and its response might represent non-consecutive entries in the history.

We can define a partial order on the operations. An operation \( e_1 \) precedes an operation \( e_2 \) \( (e_1 < e_2) \) if the response moment of \( e_1 \) comes before the invocation moment of \( e_2 \).

A sequential history is a history with a total ordering, where no operation intervals overlap (there is no operation interleaving, just consecutive invocation/response pairs).

The linearization of a history is performed in two steps. First, unfinished operations (invocations with no matching response) are removed or finished. Then, invocations and responses are reordered such that the result is a legal (in terms of the semantics of objects) sequential history and it is consistent with the operation order we had in the first place. Here, consistent means that if two operations are ordered \( (e_1 < e_2) \) in the original history, they are also ordered \( (e_1 < e_2) \) in the linearization.

Let’s consider the following example involving a queue accessed from two concurrent processes.

![Diagram of queue operations](image)

To linearize the history in the diagram above, we need to push the q.put(y) operation to the right; otherwise, if we push it to the left, the queue does not respect the FIFO semantics and the resulting ordering would not be legal.

We have the following correctness conditions:

1. sequential consistency
2. linearizability
3. serializability
4. strict serializability

In all of these, what we observe must be a sequential legal history. With respect to maintaining a sequential history, sequential consistency preserves program order, linearizability preserves operation ordering, serializability enforces contiguous transactions and strict serializability preserves
the partial ordering on transactions. If we look at transactions consisting of just one operation, there is no difference between linearizability and strict serializability.

Serializability brings the power to compose operations, to make it appear that no operation happens in between the operations that compose the transaction.

Note on contiguous transactions: if $T_1$ is a transaction consisting of operations $e_1, e_2, e_3$ and $T_2$ is another transaction consisting of operations $e_4, e_5, e_6$, then the sequential history either has to be $e_1, \ldots, e_6$ or $e_4, \ldots, e_6, e_1, \ldots, e_3$.

One downside of using linearizability for correctness is that it relies on having a precise specification of the allowed operations. Most of the time, existing software does not have precise enough specifications.

The extra power provided by (strict) serializability comes with the price of more overhead (needed for mechanisms such as two-phase locking).

Linearizability gives isolation at the level of operations, whereas serializability gives isolation at the level of transactions.