Principles of Concurrency

Lecture 8
Linearizability
Basic Idea

- Sequential consistency reasons about concurrent actions by enforcing a global ordering over thread operations
- Linearizability is a property of a concurrent program expressed in terms of time, not order
- Both definitions reduce concurrent actions to some form of sequential execution
Linearizability

- The state of an object can be manipulated by its methods concurrently.
- Two methods that do not overlap in a history are always ordered with respect to global time (happens-before relation).
- Two methods that do overlap in a history are ordered in a way that preserves the illusion that the object is sequential (i.e., not manipulable by its methods concurrently).
  - Reading an object should reflect the effect of the most recent write, regardless of the thread performing the read.

Complications

- Definition appeals to notion of global time.
- Concurrently executing methods can overlap.
Linearizability

Non-overlapping actions

- \( a.\text{write}(x) \)
- \( a.\text{read()} \)
- \( a.\text{read()} \)

Overlapping actions

- \( a.\text{write}(x) \)
- \( a.\text{read()} \)
- \( a.\text{read()} \)
- \( a.\text{read()} \)

Need to enforce:
- read returns result of most recent write
- once effect of a write becomes visible, all subsequent reads must return the value, until a new write completes

reads 0

what value can be read here if the execution was linearizable?

reads y
Example

q = new ConcurrentQueue()

T1:  T2:
q.add(10)  t = q.remove()

After execution, can assert that either

\[ t = \text{fail} \quad \&\& \quad q.\text{size} = 1 \quad \text{or} \]
\[ t = 10 \quad \&\& \quad q.\text{size} = 0 \]

q = new ConcurrentQueue()

T1:  T2:
q.add(10)  q.add(20)
t = q.remove()  t = q.remove()

After execution, can assert that:

\[ q.\text{size} = 0 \quad \&\& \quad \]
\[ (t = 10 \quad \&\& \quad u = 20) \quad \text{||} \]
\[ (t = 20 \quad \&\& \quad u = 10) \]
An object is linearizable if in all its executions, the effects of its methods appear to take place atomically at a single temporal point between the method’s call and return.
Concurrent Histories

Sequence of events comprising invocations and returns

\(<T1, q, \text{Invk}, \text{add}, 10>\>
\(<T2, q, \text{Invk}, \text{rem}>\>
\(<T1, q, \text{Ret}, \text{add}, \text{void}>\>
\(<T2, q, \text{Ret}, \text{rem}, 10>\>

Assume histories are complete - every invocation has a return

Two histories are equivalent if
- threads perform operations in the same order
- the return values they observe are the same

Two operations in a history are concurrent if their intervals overlap:
\((\text{op1.Invk} < \text{op2.Ret}) \&\& (\text{op2.Invk} < \text{op1.Ret})\)
A history is serial if every invocation is immediately followed by a return.

The set of all serial histories defines the sequential behavior of the object:

- `<T1, q, Invk, add, 10>`
- `<T1, q, Ret, add, void>`
- `<T2, q, Invk, rem>`
- `<T2, q, Ret, rem, 10>`
Linearizability

A concurrent history is linearizable if it is equivalent to a serial history in the sequential specification.

This means that all operations that take place “before” in the concurrent history also take place “before” in the serial one.