Concurrent Objects and Linearizability

Nir Shavit
Subing for N. Lynch
Fall 2003

Objectivism
• What is a concurrent object?
  – How do we describe one?
  – How do we implement one?
  – How do we tell if we’re right?

Sequential Objects
• Each object has a state
  – Usually given by a set of fields
  – Queue example: sequence of items
• Each object has a set of methods
  – Only way to manipulate state
  – Queue example: enq and deq methods
Sequential Specifications

• If (precondition)
  - the object is in such-and-such a state
  - before you call the method,
• Then (postcondition)
  - the method will return a particular value
  - or throw a particular exception.
• and (postcondition, con’t)
  - the object will be in some other state
  - when the method returns,

Pre and PostConditions for Dequeue

• Precondition:
  - Queue is non-empty
• Postcondition:
  - Returns first item in queue
• Postcondition:
  - Removes first item in queue
• You got a problem with that?

Pre and PostConditions for Dequeue

• Precondition:
  - Queue is empty
• Postcondition:
  - Throws Empty exception
• Postcondition:
  - Queue state unchanged

Why Sequential Specifications Totally Rock

• Documentation size linear in number of methods
  - Each method described in isolation
• Interactions among methods captured by side-effects on object state
  - State meaningful between method calls

Why Sequential Specifications Totally Rock (con’t)

• Can add new methods (by subclassing)
  - Without changing descriptions of old methods
• These properties are so familiar, we don’t think about them
  - But perhaps we should …

Methods Take Time
Sequential vs Concurrent

- Sequential:
  - Methods take time? Who knew?
- Concurrent:
  - Method call is not an event
  - Method call is an interval.

Concurrent Methods Take Overlapping Time

Sequential vs Concurrent

- Sequential:
  - Object needs meaningful state only between method calls
- Concurrent:
  - Because method calls overlap, object might never be between method calls

Sequential vs Concurrent

- Sequential:
  - Each method described in isolation
- Concurrent:
  - Must characterize all possible interactions with concurrent calls
  - What if two enq overlap?
  - Two deq? enq and deq? ...

Sequential vs Concurrent

- Sequential:
  - Can add new methods without affecting older methods
- Concurrent:
  - Everything can potentially interact with everything else

The High-Order Bit

- What does it mean for a concurrent object to be correct?
- What is a concurrent FIFO queue?
  - FIFO means strict temporal order
  - Um, like, that’s why they call it “FIFO”
  - Concurrent means ambiguous temporal order
Linearizability Manifesto

• Each method should
  - "take effect"
  - Instantaneously
  - Between invocation and response events

• Any such concurrent object is
  - Linearizable™

Comments on Manifesto

• Common Sense, not Science
• Scientific justification:
  - Facilitates reasoning
  - Nice mathematical properties
• Common-sense justification
  - Preserves real-time order
  - Matches my intuition (sorry about yours)

More Comments on Manifesto

• Proposed in 1990
  - Since then accepted almost anywhere
  - Don't leave home without it …
• Not universally adopted
  - Usually for good, but specialized reasons
  - But most feel need to justify why not

Reasoning

• People are
  - OK with sequential reasoning
  - Challenged by concurrent reasoning
    • Air traffic control
    • Toddler room in day-care center
• Most concurrent models
  - Propose some kind of equivalence
  - Make concurrent problems sequential
  - Except they sometimes do it wrong …

Concurrent Specifications

• Naïve approach: world of pain
  - We must specify unspeakable number of possible multi-way interactions
• Linearizability: same as it ever was
  - Methods still described by pre- and postconditions

Example

\[
\begin{align*}
&\text{time} \\
&q.\text{enq}(x) \\
&q.\text{enq}(y) \\
&q.\text{deq}(x) \\
&q.\text{deq}(y)
\end{align*}
\]
Example

\[\text{q.enq}(x) \quad \text{q.enq}(y) \quad \text{q.deq}(y)\]

\textbf{not linearizable}

Example

\[\text{q.enq}(x) \quad \text{q.deq}(x) \quad \text{q.enq}(x) \quad \text{q.deq}(x)\]

\textbf{linearizable}

Example

\[\text{q.enq}(x) \quad \text{q.enq}(y) \quad \text{q.deq}(y) \quad \text{q.deq}(x)\]

Comme ci

Example

\[\text{q.enq}(x) \quad \text{q.enq}(y) \quad \text{q.deq}(y) \quad \text{q.deq}(x)\]

Comme ça

Example

\[\text{q.enq}(x) \quad \text{q.enq}(y) \quad \text{q.deq}(y) \quad \text{q.deq}(x)\]

\textbf{multiple orders OK}
Read/Write Variable Example

not linearizable

write(0)  read(1)  write(2)  write(1)

write(1) already happened

time

read(0)  write(1)

write(1) already happened

Write(2)

not linearizable

read(1)  write(1)  write(2)

Linearizable

write(0)  read(1)  write(2)

write(1)  read(1)

time

read(1)  write(1)  write(2)

Linearizable

Split Method Calls into Two Events

• Invocation
  - method name & args
  - q.enq(x)

• Response
  - result or exception
  - q.enq(x) returns void
  - q.deq() returns x
  - q.deq() throws empty

Formal Model

• Define precisely what we mean
  - Ambiguity is bad when intuition is weak

• Allow reasoning
  - Formal
  - But mostly informal
    - In the long run, actually more important
    - Ask me why!
Invocation Notation

A q.enq(x)

thread method arguments

Response Notation

A q: void

thread object result

Response Notation (cont)

A q: empty()

thread exception object

History

A q.enq(3)
A q: void
A q.enq(5)
B p.enq(4)
B p: void
B q.deq()
B q: 3

Sequence of invocations and responses

Definition

• Invocation & response match if

Thread names agree Object names agree

Method call

Object Projections

A q.enq(3)
A q: void
B p.enq(4)
B p: void
B q.deq()
B q: 3
Object Projections

A q.enq(3)
A q: void
H|q =
B q.deq()
B q: 3

Thread Projections

A q.enq(3)
A q: void
H =
B p.enq(4)
B p: void
B q.deq()
B q: 3

Thread Projections

H|B =
B p.enq(4)
B p: void
B q.deq()
B q: 3

Complete Subhistory

A q.enq(3)
A q: void
A q.enq(5)
H =
B p.enq(4)
B p: void
B q.deq()
B q: 3

An invocation is pending if it has no matching response

Complete Subhistory

May or may not have taken effect

Complete Subhistory

Complete(H) =
B p.enq(4)
B p: void
B q.deq()
B q: 3

discard pending invocations

Complete Subhistory


Sequential Histories

A q.enq(3)
A q:enq(3)
match

A q:enq(3)
A q: void

B p.enq(4)
B p:enq(4)
match

B p: enq(4)
B p: void

B q.deq()
B q: deq()
match

B q: deq() 
B q:3

B q:3

Final pending invocation OK

Well-Formed Histories

A q.enq(3)
A q:enq(3)

A q:enq(3)
A q: void

B p.enq(4)
B p:enq(4)

B p: enq(4)
B p: void

B q.deq()
B q: deq()

B q: deq() 
B q:3

B q:3

H = B p.enq(4)
B p: enq(4)
B q: deq() 
B q:3

Well-Formed Histories

Per-thread projections sequential

H = B p.enq(4)
B p: enq(4)

A q.enq(3)
A q: enq(3)

A q: enq(3)
A q: void

B p.enq(4)
B p: enq(4)

B p: enq(4)
B p: void

B q.deq()
B q: deq()

B q: deq() 
B q:3

A q: void

B q:3

H|B =

H|A = A q.enq(3)
A q: void

A q.enq(3)
A q: void

 Equivalent Histories

Threads see the same thing in both

\[
\begin{align*}
H & = A q.enq(3) \\
H & = A q: void \\
B & = B p.enq(4) \\
B & = B p: void \\
G & = B q.deq() \\
G & = B q:3 \\
H | A & = G | A \\
H | B & = G | B
\end{align*}
\]

Sequential Specifications

• A sequential specification is some way of telling whether a
  • Single-thread, single-object history
  • Is legal
• My favorite way is using
  • Pre and post-conditions
  • But plenty of other techniques exist ...

Legal Histories

• A sequential (multi-object) history H is legal if
  • For every object x
  • H|x is in the sequential spec for x
**Method Call**

A q.enq(3)  
B p.enq(4)  
B p.void  
B q.deq()  
A q: void  
B q: 3

Interval between invocation and response events

**Precedence**

A q.enq(3)  
B p.enq(4)  
B p.void  
A q: void  
B q.deq()  
B q: 3

A method call precedes another if response event precedes invocation event

**Non-Precedence**

A q.enq(3)  
B p.enq(4)  
B p.void  
B q.deq()  
A q: void  
B q: 3

Some method calls overlap one another

**Notation**

- Given
  - History H
  - method executions $m_0$ and $m_1$ in H
- We say $m_0 \rightarrow_H m_1$, if
  - $m_0$ precedes $m_1$
- Relation $m_0 \rightarrow_H m_1$ is a
  - Partial order
  - Total order if H is sequential

**Linearizability**

- History H is **linearizable** if it can be extended to $G$ by
  - Appending zero or more responses to pending invocations
  - Discarding other pending invocations
- So that $G$ is equivalent to
  - Legal sequential history $S$
  - where $\rightarrow_G \subseteq \rightarrow_S$

**Remarks**

- Some pending invocations
  - Took effect, so keep them
  - Discard the rest
- Condition $\rightarrow_G \subseteq \rightarrow_S$
  - Means that $S$ respects "real-time order" of $G"
Example

A q.enq(3)
B q.enq(4)
B q: void
B q.deq()
B q: 4
B q.enq(6)

Example

A q.enq(3)
B q.enq(4)
B q: void
B q.deq()
B q: 4
B q.enq(6)
A q: void

Example

A q.enq(3)
B q.enq(4)
B q: void
B q.deq()
B q: 4
A q: void

Example

A q.enq(3)
B q.enq(4)
B q: void
B q.deq()
B q: 4
A q: void

Example

A q.enq(3)
B q.enq(4)
B q: void
B q.deq()
B q: 4
A q: void

Example

A q.enq(3)
B q.enq(4)
B q: void
B q.deq()
B q: 4
A q: void

Equivalent sequential history
Concurrency

• How much concurrency does linearizability allow?
• When must a method invocation block?

Focus on total methods
- Defined in every state
• Example:
  - `deq()` that throws `Empty` exception
  - Versus `deq()` that waits …
• Why?
  - Otherwise, blocking unrelated to synchronization

Concurrency

• Question: When does linearizability require a method invocation to block?
• Answer: never.
• Linearizability is non-blocking

Non-Blocking Theorem

If method invocation
A q.inv(…)
is pending in history H, then there
exists a response
A q:res(…)
such that
H + A q:res(…)
is linearizable

Proof

• Pick linearization S of H
• If S already contains
  - Invocation A q.inv(…)
  - Then we are done.
• Otherwise, pick a response such that
  - S + A q.inv(…) + A q:res(…)
  - Possible because object is total.

Locality Theorem

• History H is linearizable if and only if
  - For every object x
  - H[x] is linearizable
• We care about objects only!
  - (Materialism anyone?)
Why Does Locality Matter?

- Modularity
- Can prove linearizability of objects in isolation
- Can compose independently-implemented objects

Reasoning About Linearizability

public class Queue {
    int head = 0, tail = 0;
    Object[QSIZE] items;
    public synchronized void enq(Object x) {
        while (this.tail-this.head == QUEUE_SIZE)
            this.wait();
        this.items[tail++ % QUEUE_SIZE] = x;
        this.notifyAll();
    }
}

Now for something completely different

- Let's try the same thing without mutual exclusion
- For simplicity, only two threads please

More Reasoning

public class LockFreeQueue {
    int head = 0, tail = 0;
    Item[QSIZE] items;
    public void enq(Item x) {
        while (tail-head == QSIZE); // busy-wait
        items[tail % QSIZE] = x; tail++;
    }
    public Item deq() {
        while (tail == head); // busy-wait
        Item item = items[head % QSIZE]; head++;
        return item;
    }
}
More Reasoning

```
public class LockFreeQueue {
    int head = 0, tail = 0;
    Item[QSIZE] items;
    public void enq(Item x) {
        while (tail-head == QSIZE); // busy-wait
        items[tail % QSIZE] = x; tail++;
    }
    public Item deq() {
        while (tail == head);     // busy-wait
        Item item = items[head % QSIZE]; head++;
        return item;
    }
}
```

Strategy

- Identify one atomic step where method "happens"
  - Critical section
  - Machine instruction
- Doesn't always work
  - In theory
  - Usually works in practice

Alternative: Sequential Consistency

- History equivalent to some sequential history
- No need to preserve real-time order
- Often used to describe multiprocessor memory architectures

Theorem

Sequential Consistency is not a local property
H|q Sequentially Consistent

Ordering imposed by p

Ordering imposed by q

Combining Orderings

Not Sequentially Consistent

Serializability

- A transaction is a finite sequence of method calls
- It is serializable if
  - transactions appear to execute serially
- Strictly serializable if
  - order is compatible with real-time
- Used in databases
Serializability is Blocking

\[
\begin{align*}
&x.\text{read}(0) & \quad & y.\text{read}(0) \\
&x.\text{write}(1) & \quad & y.\text{write}(1)
\end{align*}
\]

Serializability not Local

- Cannot mix, say
  - Two-phase locking
  - Timestamp synchronization
- May serialize transactions at different objects in opposite orders

Comparison

- Serializability appropriate for
  - Fault-tolerance
  - Multi-step transactions
- Linearizability appropriate for
  - Single objects
  - Multiprocessor synchronization

Critical Sections

- Easy way to implement linearizability
  - Take sequential object
  - Make each method a critical section
- Like synchronized methods in Java™
- Problems
  - Blocking
  - No concurrency

Summary

- Linearizability
  - Operation takes effect instantaneously between invocation and response
- Uses sequential specification
  - No \(O(n^2)\) interactions

Summary

- Non-Blocking
  - Never required to pause method call
- Locality
  - Can verify linearizability per object
  - Can compose correctly
- Granularity matters
Fact

- Any partial order
  - Never required to pause method call
- Locality
  - Can verify linearizability per object
  - Can compose correctly
- Granularity matters