Sequential Consistency and Linearizability

(Or, Reasoning About Concurrent Objects)

Acknowledgement:
Slides partially adopted from the companion slides for the book "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit
What We'll Cover Today

Chapter 3 of:

The Art of Multiprocessor Programming

Digital copy can be obtained via WUSTL library:
http://catalog.wustl.edu/search/
Concurrent Computation

memory

object

object
Objectiveism

What does it mean for a concurrent object to be correct?

- How do we specify its behavior?
- How do we implement one?
- How do we tell if the implementation is correct?

Both **sequential consistency** and **linearizability** are correctness conditions for concurrent objects.
Sequential Consistency [1]

• Typically used as a *memory consistency model*, which specifies in what order the memory operations may appear to execute.

• You can think of a memory location as a concurrent object.
Sequential Consistency [1]

“The result of any execution is the same as if the operations of all the processors* were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.”

— Leslie Lamport, 1979

• The sequence of instructions from threads executing concurrently are **interleaved** to form a **global linear order** of all instructions.
• The sequence of instructions from a given thread is defined by the thread’s program.
• Implicit: each memory location is **coherent** according to the global linear order.

* I will be using processor vs thread interchangeably.
What SC Buys Us

Relating to the dag:
Under SC, the final result we get from executing this dag is as if the execution followed some topological sort of the dag.

<table>
<thead>
<tr>
<th>r1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>r2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>possible? (example)</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>Not under SC</td>
</tr>
<tr>
<td></td>
<td>(2,4,3,5)</td>
<td>(2,3,4,5)</td>
<td>(4,5,2,3)</td>
<td>SC</td>
</tr>
</tbody>
</table>

SC precludes executions that don't conform to program order.
public void lock() {
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i) {};
}

public void unlock() {
    flag[i] = false;
}
Crux of Peterson Proof Needs SC

(1) \( \text{write}_B(\text{flag}[B]=\text{true}) \implies \text{write}_B(\text{victim}=B) \)

(3) \( \text{write}_B(\text{victim}=B) \implies \text{write}_A(\text{victim}=A) \)

(2) \( \text{write}_A(\text{victim}=A) \implies \begin{align*} &\text{read}_A(\text{flag}[B]) \\ &\text{read}_A(\text{victim}) \end{align*} \)

A read flag[B] == true and victim == A, so it could not have entered the CS (QED)

The proof assumes execution follows program order, and that each memory location is coherent.
Linearizability [2]

• Originally developed to reason about behaviors of concurrent objects.
• Stronger than sequential consistency (we will say more about that later).
public class FIFOQueue {
    int head = 0, tail = 0;
    items = (T[]) new Object[capacity];

    public void enq(T x) {
        if (tail-head == capacity)
            throw new FullException();
        items[tail % capacity] = x;
        tail++;
    }

    public T deq() {
        if (tail == head)
            throw new EmptyException();
        Item item = items[head % capacity];
        head++;
        return item;
    }
}
Now Consider the Following Scenario

• The FIFO queue is accessed by two threads concurrently.
• But,
  – One thread \texttt{enq only}
  – The other \texttt{deq only}
Wait-free 2-Thread Queue

deq()

enq(z)

head

tail

z
Wait-free 2-Thread Queue

head

result = x

queue[tail] = z
Wait-free 2-Thread Queue
public class FIFOQueue {
    int head = 0, tail = 0;
    items = (T[]) new Object[capacity];

    public void enq(T x) {
        if (tail-head == capacity)
            throw new FullException();
        items[tail % capacity] = x;
        tail++;
    }

    public T deq() {
        if (tail == head)
            throw new EmptyException();
        Item item = items[head % capacity];
        head++;
        return item;
    }
}
What *is* a Concurrent Queue?

• Need a way to specify a concurrent queue object
• Need a way to prove that an algorithm implements the object’s specification
• Let’s talk about object specifications …
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 Deq

• Precondition:
  – Queue is non-empty

• Postcondition:
  – Returns first item in queue

• Postcondition:
  – Removes first item in queue
Pre and PostConditions for Deq

• Precondition:
  – Queue is empty

• Postcondition:
  – Throws Empty exception

• Postcondition:
  – Queue state unchanged
Why Sequential Specifications Totally Rock

• Interactions among methods captured by side-effects on object state
  – State meaningful between method calls
• Documentation size linear in number of methods
  – Each method described in isolation
• Can add new methods
  – Without changing descriptions of old methods
Sequential vs Concurrent

• Sequential
  – Methods take time? Who knew?
  – Objects need meaningful states only *between* method calls.

• Concurrent
  – Method call is not an event
  – Method call is an interval.
  – Because method calls overlap, object might *never* be between method calls.
Sequential vs Concurrent

**Sequential:**
- Each method can be described in isolation.
- Can add new methods without affecting older methods

**Concurrent:**
- Must characterize *all* possible interactions with concurrent calls
  - What if two `enq()` calls overlap?
  - Two `deq()` calls? `enq()` and `deq()`? …
- Everything can potentially interact with everything else
The Big Question

• What does it mean for a concurrent object to be correct?
  – What is a concurrent FIFO queue?
  – FIFO means strict temporal order
  – Concurrent means ambiguous temporal order
Intuitively...

```java
public T deq() throws EmptyException {
    lock.lock();
    try {
        if (tail == head)
            throw new EmptyException();
        T x = items[head % items.length];
        head++;
        return x;
    } finally {
        lock.unlock();
    }
}
```

All queue modifications are mutually exclusive
Let's capture the idea of describing the concurrent via the sequential.

Behavior is “Sequential”
Linearizability [2]

• Each method should
  – “take effect” instantaneously (linearization point)
  – between invocation (making a method call) and response (returning from a method call) events

• If we can look at the trace from parallel execution, and pick a linearization point for all method invocations such that the history "make sense" (i.e., matches sequential behavior), then the history is linearizable.

• A linearizable object: one all of whose possible executions are linearizable.
Example

q.enq(x)
q.deq(x)
q.enq(y)
q.deq(y)
Example

q.enq(x)
q.enq(y)
q.deq(x)
q.deq(y)
q.enq(x)
q.enq(y)
q.deq(x)
q.deq(y)

linearizable
Example

```
quenq(x)
quenq(y)
qudeq(x)
qudeq(y)
quenq(x)
quenq(y)
qudeq(x)
qudeq(y)
```
Example

not linearizable
Example

Multiple orders OK linearizable

q.enq(x)

q.enq(y)

q.deq(y)

q.deq(x)

time
Read/Write Register Example

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

write(1) already happened

read(0)
Read/Write Register Example

write(0) -> read(1) -> write(2)
write(1) already happened

not linearizable

read(0)
Read/Write Register Example

- write(0)
- read(1)
- write(2)
- write(1) already happened
- read(1)
Read/Write Register Example

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

write(1) already happened

not linearizable
Read/Write Register Example

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

time
Read/Write Register Example

linearizable

write(0)

write(1)

write(2)

read(1)

time
Talking About Executions

• Why?
  – Can’t we specify the linearization point of each operation without describing an execution?

• Not Always
  – In some cases, linearization point depends on the execution
Linearizability, Formally

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`
Invocation Notation

A q.enq(x)

thread

object

method

arguments
Response Notation

Method is implicit

A q: void

thread

object

result
Response Notation

Method is implicit

A q: empty()
Definition

- Invocation & response *match* if

  - Thread names agree
  - Object names agree

  A q.enq(3) q: void
History: Describing an Execution

\[ H = \]

\begin{align*}
A &\text{ q.enq(3)} \\
A &\text{ q: void} \\
A &\text{ q.enq(5)} \\
B &\text{ p.enq(4)} \\
B &\text{ p: void} \\
B &\text{ q.deq()} \\
B &\text{ q: 3}
\end{align*}

Sequence of invocations and responses
Object Projection

H|q =

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

Picking out entries performed on the target object.
Thread Projection

H|B =

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

Picking out entries performed by a given thread.
History: Describing an Execution

An invocation is **pending** if it has no matching response.
Each per-thread projection contains matching invocation and responses, except for the last pending invocation.
Equivalent Histories

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

\[ G = \]
- A\ q.enq(3)
- B\ p.enq(4)
- A\ q:void
- B\ p:void
- B\ q.deq()
- B\ q:3
- A\ q.enq(5)

Threads see the same thing in both

\[ H|A = G|A \]
\[ H|B = G|B \]
Sequential Specifications

• A *sequential specification* is some way of telling whether a single-thread, single-object history is legal

• For example:
  – Pre and post-conditions
  – But plenty of other techniques exist …
Legal Histories

• A sequential (multi-object) history $H$ is *legal* if
  – For every object $x$
  – $H|_{x}$ follows the sequential specification for $x$
Precedence Ordering

A \ q.\text{enq}(3)
B \ p.\text{enq}(4)
B \ p.\text{void}
A \ q.\text{void}
B \ q.\text{deq}()
B \ q:3

A method call \textit{precedes} another if response event precedes invocation event

A \ q.\text{enq}(3)
B \ p.\text{enq}(4)
B \ p.\text{void}
B \ q.\text{deq}()
A \ q.\text{void}
B \ q:3

Otherwise they \textit{overlap}
Nota&on

• Given
  – History \(H\)
  – method executions \(m_0\) and \(m_1\) in \(H\)

• We say \(m_0 \rightarrow_H m_1\), if
  – \(m_0\) precedes \(m_1\) in \(H\)

• Relation \(m_0 \rightarrow_H m_1\) is a
  – Partial order
  – Total order if \(H\) is sequential
Linearizability [2]

• History $H$ is \textit{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
  – a legal sequential history $S$
  – where $\rightarrow_G \subseteq \rightarrow_S$

Fix the invocations that already took effect and discard the rest that haven't.
Linearizability [2]

• 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
  – a legal sequential history S
  – where $\Rightarrow_G \subseteq \Rightarrow_S$

Equivalent to S captures the fact that input/output of the methods must be the same as in sequential execution.
**Linearizability [2]**

- 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
  - a legal sequential history S
  - where $\Rightarrow_G \subseteq \Rightarrow_S$

The total order in S must respect the partial order ("real-time" ordering) in the original history.
Ensuring $\to_G \subseteq \to_S$

$\to_G = \{a \to c, b \to c\}$

$\to_S = \{a \to b, a \to c, b \to c\}$

A limitation on the Choice of $S$!
Example

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

Complete this pending invocation
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

Complete this pending invocation
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

Discard this pending invocation
Example

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

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

Equivalent sequential history
Composability Theorem

• History $H$ is linearizable if and only if
  – For every object $x$
  – $H|x$ is linearizable

• Why Does Composability Matter?
  – Modularity
  – Can prove linearizability of objects in isolation
  – Can compose independently-implemented objects
Reasoning About Linearizability:
Wait-Free Concurrent 2-Thread Queue

```java
public class FIFOQueue {
    int head = 0, tail = 0;
    items = (T[]) new Object[capacity];

    public void enq(T x) {
        if (tail-head == capacity)
            throw new FullException();
        items[tail % capacity] = x;
        tail++;
    }

    public T deq() {
        if (tail == head)
            throw new EmptyException();
        Item item = items[head % capacity];
        head++;
        head++;
        return item;
    }
}
```
General Strategy

• Identify one atomic step where method “happens”
  – Critical section
  – Machine instruction

• Not necessarily a single linearization point
  – Might need to define several different ones for a given method
Linearizability: Summary

- Powerful specification tool for shared objects
- Allows us to capture the notion of objects being “atomic”
- Don’t leave home without it
Alternative: Sequential Consistency

• History H is *sequentially consistent* 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
  – a legal sequential history S

\[ G \subseteq S \]

where \( G \subseteq S \)

G is equivalent to S, which implies that S preserves program order within a thread.

Allows reordering operations by different threads to establish a global sequential order.
SC is Weaker than Linearizability

\[q.\text{enq}(x)\]
\[q.\text{enq}(y)\]
\[q.\text{deq}(y)\]
\[q.\text{enq}(y)\]
SC is Weaker than Linearizability

Yet Sequentially Consistent
Theorem

Sequential Consistency is not composable
FIFO Queue Example

\[
p.\text{enq}(x) \quad q.\text{enq}(x) \quad p.\text{deq}(y) \\
q.\text{enq}(y) \quad p.\text{enq}(y) \quad q.\text{deq}(x)
\]
H|p Sequentially Consistent

Diagram:
- p.enq(x)
- q.enq(x)
- p.deq(y)
- q.enq(y)
- p.enq(y)
- q.deq(x)

Time Arrow: time
H|q Sequentially Consistent

\begin{align*}
\text{p.enq}(x) & \quad \text{q.enq}(x) & \quad \text{p.deq}(y) \\
\text{q.enq}(y) & \quad \text{p.enq}(y) & \quad \text{q.deq}(x)
\end{align*}

time
Ordering imposed by p

```
p.enq(x)
q.enq(x)
p.deq(y)
q.enq(y)
p.enq(y)
q.deq(x)
```
Ordering imposed by q

\[ p\text{-}enq(x) \rightarrow q\text{-}enq(x) \rightarrow p\text{-}deq(y) \rightarrow q\text{-}enq(y) \rightarrow p\text{-}enq(y) \rightarrow q\text{-}deq(x) \]

time
Ordering imposed by both

\[ p\text{.enq}(x) \]
\[ q\text{.enq}(x) \]
\[ p\text{.deq}(y) \]
\[ q\text{.enq}(y) \]
\[ p\text{.enq}(y) \]
\[ q\text{.deq}(x) \]
Combining orders

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

\[ p.\text{enq}(y) \]
\[ q.\text{enq}(y) \]
\[ p.\text{enq}(y) \]

\text{time}
Hardware Memory Consistency

- No modern-day processor implements sequential consistency.
- Hardware actively reorders instructions.
- Compilers may reorder instructions, too.
- Why?
- Because most of performance is derived from a single thread’s unsynchronized execution of code.
Memory Barriers (Fences)

• A *memory barrier* (or *memory fence*) is a hardware action that enforces an ordering constraint between the instructions before and after the fence.
• A memory barrier can be issued explicitly as an instruction (e.g., in x86: mfence)
• The typical cost of a memory fence is comparable to that of an L2-cache access.
Memory Consistency in High-Level Language

• In Java, can ask compiler to keep a variable up-to-date by declaring it \textit{volatile}:
  – Adds a memory barrier after each store
  – Inhibits reordering, removing from loops, \& other “compiler optimizations”

• C++11 standard offers atomic variables that come with a set of different memory ordering constraints.
Summary: Real-World

• Hardware weaker than sequential consistency
• Can get sequential consistency at a price
• Linearizability better fit for high-level software
References


This work is licensed under a Creative Commons Attribution-ShareAlike 2.5 License.

- You are free:
  - to Share — to copy, distribute and transmit the work
  - to Remix — to adapt the work
- Under the following conditions:
  - Attribution. You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
  - Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  - http://creativecommons.org/licenses/by-sa/3.0/.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.