Today’s agenda

1. concurrent objects and correctness

2. quiescent and sequential consistency

3. linearizability
Terminology: concurrent objects

- a **concurrent object** is a data object shared by concurrent processes

  \(\Rightarrow\) has a type defining possible values, and primitive methods that provide the only means of creation/manipulation

  \(\Rightarrow\) e.g. a shared data structure, a shared message queue, ...

- a **concurrent system** is a collection of sequential processes that communicate through concurrent objects
Specifying correctness of operations

• in a **sequential system**, it is easy to specify the behaviour of methods

  => *pre* and *post*conditions

  \[ \{\text{pre}\} \ q.\text{op} \ \{\text{post}\} \]

  => *methods cannot be called on objects that are in an “intermediate state”*

• in a **concurrent system**, need to accommodate interleavings of method invocations
What does it mean for concurrent objects to be correct?

• typically boils down to some notion of equivalence with sequential behaviour

• consider a simple, lock-based concurrent FIFO queue
What does it mean for concurrent objects to be correct?

- typically boils down to some notion of equivalence with sequential behaviour

- consider a simple, lock-based concurrent FIFO queue

```java
public void enq(int x) throws FullException {
  lock.lock();
  try {
    if <<queue full>>
      { throw new FullException(); }
    <<enqueue x>>
  } finally {
    lock.unlock();
  }
}

public int deq() throws EmptyException {
  lock.lock();
  try {
    if <<queue empty>>
      { throw new EmptyException(); }
    <<dequeue x>>
  } finally {
    lock.unlock();
  }
}```
A lock-based concurrent FIFO queue

A

q.enq(a)

B

q.enq(b)

C

q.deq(b)
A lock-based concurrent FIFO queue

A lock() <<enq a>> unlock()

B lock() <<enq b>> unlock()

C lock() ! unlock() lock() <<deq b>> unlock()
A lock-based concurrent FIFO queue

FIFO Queue Timeline
A lock-based concurrent FIFO queue

**Timeline**

**A**
- lock()
- «enq a>>
- unlock()

**B**
- lock()
- «enq b>>
- unlock()

**C**
- lock()
- «deq b>>
- unlock()
A lock-based concurrent FIFO queue

A

lock() <<enq a>>

unlock()

B

lock() <<enq b>> unlock()

C

lock() unlock() <<deq b>>

FIFO Queue Timeline

<<enq b>>
A lock-based concurrent FIFO queue

Timeline
A lock-based concurrent FIFO queue

Timeline:

A: lock() <<enq a>> unlock()
B: lock() <<enq b>> unlock()
C: lock() <<deq b>> unlock()

FIFO Queue Timeline:

<<enq b>> <<enq a>> <<deq b>>
What happens if we drop the locks?

```java
public void enq(int x) throws FullException {
    lock.lock();
    try {
        if (queue.full) {
            throw new FullException();
        }
        enqueue x;
    } finally {
        lock.unlock();
    }
}

public int deq() throws EmptyException {
    lock.lock();
    try {
        if (queue.empty) {
            throw new EmptyException();
        }
        dequeue x;
    } finally {
        lock.unlock();
    }
}
```
What happens if we drop the locks?

```
public void enq(int x) throws FullException {
    lock.lock();
    try {
        if <<queue full>> {
            throw new FullException();
        }
        <<enqueue x>>
    } finally {
        lock.unlock();
    }
}
```

```
public int deq() throws EmptyException {
    lock.lock();
    try {
        if <<queue empty>> {
            throw new EmptyException();
        }
        <<dequeue x>>
    } finally {
        lock.unlock();
    }
}
```

are there circumstances in which this queue can be correct?

what does “correct” mean?
Reasoning about concurrent objects: a principle

- concurrent objects may have methods with finer-grained locking or no locking at all

- need to be able to specify and reason about implementations without relying on method-level locking

- but the example illustrates an important principle:

  it’s easier to reason about concurrent objects if we can map their concurrent executions to sequential ones
Which “equivalences” with sequential behaviour do we care about?

- do we care about program order, fairness, ...?

- in practice, different applications require different “strengths” of correctness conditions

  => print job queue for a lightly loaded printer
  => banking server (e.g. transfer money from savings; withdraw £50)
  => stock-trading server
We will consider three correctness conditions

**quiescent consistency**

\[ \Rightarrow \text{ whenever an object becomes quiescent, then the execution so far is equivalent to some sequential execution of the completed calls } \]

**sequential consistency**

\[ \Rightarrow \text{ method calls should appear to take effect in a sequential order consistent with the program order } \]

**linearizability**

\[ \Rightarrow \text{ each method call should appear to take effect instantaneously at some moment between its invocation and response } \]
Next on the agenda

1. concurrent objects and correctness

2. quiescent and sequential consistency

3. linearizability
Terminology: method calls

- individual threads sequentially execute method calls that have *invocation* and *response* events

- a method is *pending* if its call has occurred, but not its response
What conditions / restrictions do we need?

• let’s derive some principles from examples of unacceptable behaviours
What conditions / restrictions do we need?

• let’s derive some principles from examples of unacceptable behaviours
What conditions / restrictions do we need?

- let’s derive some principles from examples of unacceptable behaviours

A

r.write(7)

B

r.write(-3)  r.read(-7)

one might expect to read 7 or -3, not a mixture of both
A principle

method calls should appear to happen in a one-at-a-time, sequential order
A principle

method calls should appear to happen in a one-at-a-time, sequential order
A principle

Method calls should appear to happen in a one-at-a-time, sequential order.

A

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>r.write(7)</td>
<td></td>
</tr>
</tbody>
</table>

B

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>r.write(-3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>r.read(-3)</td>
</tr>
</tbody>
</table>

A green check mark indicates that the calls are in the correct order.
A principle

Method calls should appear to happen in a one-at-a-time, sequential order!

Too weak alone!

Permits, e.g., readers to always return the object’s initial state.
A principle

method calls should appear to happen in a one-at-a-time, sequential order

A

r.write(7)  

B

r.write(-3)

r.read(???)
A principle

method calls should appear to happen in a one-at-a-time, sequential order

A

```
r.write(7)
```

B

```
r.write(-3)
```

```
r.read(???)
```

*it would be unacceptable to read 7 here*
Quiescent consistency

method calls should appear to happen in a one-at-a-time, sequential order

+ 

method calls separated by a period of quiescence should appear to take effect in their real-time order

NB: an object is quiescent if it has no pending method calls
Quiescent consistency

Quiescence

A

r.write(7)

B

r.write(-3)

r.read(-3)

quiescence  quiescence
Quiescent consistency

A

r.write(7)

B

r.write(-3)

r.read(-3)

quiescence
Quiescent consistency

A
r.write(7)

B
r.write(-3)

r.read(7)

quiescence
What if program order matters?

- should this behaviour be allowed?  \textit{(example by Huisman)}
What if program order matters?

• should this behaviour be allowed?  (example by Huisman)
What if program order matters?

- should this behaviour be allowed?  (example by Huisman)

<table>
<thead>
<tr>
<th>A</th>
<th>r.read(?)</th>
<th>r.read(?)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>r.write(7)</td>
<td>r.write(-3)</td>
</tr>
</tbody>
</table>

not acceptable: the value read is not the last it wrote
Sequential consistency

- method calls should appear to happen in a one-at-a-time, sequential order
- method calls should appear to take effect in program order
Sequential consistency

- method calls should appear to happen in a one-at-a-time, sequential order

+ method calls should appear to take effect in program order

i.e. in any concurrent execution, there is a way to order the method calls sequentially so that they are (1) consistent with program order; and (2) meet the object’s sequential specification
Sequential consistency

A

r.read(?)  r.read(?)

B

r.write(7)  r.write(-3)  r.read(-3)
Is this execution sequentially consistent?

A

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

B

q.enq(y)  q.deq(x)
Is this execution sequentially consistent?

A

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

B

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

yes! two possible consistent orderings
Is this execution sequentially consistent?

A: enqueues x
B: enqueues y
B: dequeues x
A: dequeues y

yes! two possible consistent orderings
Is this execution sequentially consistent?

A

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

B

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

yes! two possible consistent orderings

A: enqueues x  B: enqueues y
B: enqueues y  A: enqueues x
B: dequeues x  A: dequeues y
A: dequeues y  B: dequeues x
Is this execution sequentially consistent?

A: enqueues x
B: enqueues y
B: dequeues x
A: dequeues y

B: enqueues y
A: enqueues x
A: dequeues y
B: dequeues x

sequential program order preserved!

yes! two possible consistent orderings
Is this execution sequentially consistent? (from Huisman)

A

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

B

q.deq(y)  q.deq(x)
Is this execution sequentially consistent?
(from Huisman)

A

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

B

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

X proof?
Quiescent vs. sequential consistency

- quiescent and sequential consistency are incomparable
  
  => quiescent consistency does not necessarily preserve program order  
  => sequential consistency is unaffected by quiescent periods

- a correctness condition C is compositional if whenever every object satisfies C, the system as a whole satisfies C

  => quiescent consistency is compositional  
  => sequential consistency, unfortunately, is not compositional
Next on the agenda

1. concurrent objects and correctness
2. quiescent and sequential consistency
3. linearizability
Strengthening sequential consistency to gain compositionality

- should this (sequentially consistent) behaviour be allowed?
Linearizability

each method call should appear to take effect instantaneously at some moment between its invocation and response

an object is linearizable if all of its possible executions are linearizable
Linearizability

Each method call should appear to take effect instantaneously at some moment between its invocation and response.
Linearizability

Each method call should appear to take effect instantaneously at some moment between its invocation and response.

A

q.enq(2) — q.deq(5)

B

q.enq(5)

X sequentially consistent, but not linearizable
Linearizability

Each method call should appear to take effect instantaneously at some moment between its invocation and response.

A
q.enq(2) → q.deq(2)

B
q.enq(5)

q
Linearizability

Each method call should appear to take effect instantaneously at some moment between its invocation and response.
Linearizability: examples

A

q.enq(2)  q.deq(5)

B

q.enq(5)  q.deq(2)
Linearizability: examples

A

\[ q \text{.enq}(2) \]
\[ q \text{.enq}(5) \]
\[ q \text{.deq}(2) \]
\[ q \text{.deq}(5) \]

B

\[ q \text{.enq}(5) \]
\[ q \text{.deq}(2) \]

q

\[ q \text{.enq}(2) \]
\[ q \text{.enq}(5) \]
\[ q \text{.deq}(2) \]
\[ q \text{.deq}(5) \]

linearizable
Linearizability: examples

A  
- r.write(0)  
- r.read(1)  
- r.write(2)  

B  
- r.write(1)  
- r.read(0)
Linearizability: examples

A: r.write(0)  r.read(1)  r.write(2)
B: r.write(1)  r.read(0)

X  not sequentially consistent, not linearizable
Linearizability: examples

A

\[ r.\text{write}(0) \quad r.\text{write}(2) \]

B

\[ r.\text{write}(1) \quad r.\text{read}(1) \]
Linearizability: examples

```
A
r.write(0)
r.write(2)

B
r.write(1)
r.read(1)

r
r.write(0)
r.write(1)
r.read(1)
r.write(2)
```

linearizable
Linearization points

• to show that a concurrent object is linearizable, one must identify for each method a linearization point where the method takes effect

• for lock-based objects, these are the critical sections

• for lock-free approaches, the linearization point is a single step where they effects of the method call become visible to other method calls
Sequential consistency vs. linearizability

- Linearizable executions are also sequentially consistent.

- Sequential consistency is less restrictive: allows method calls to take effect after their response.

- Linearizability is compositional: the result of composing linearizable objects is linearizable.
Formal definitions

• a call of an operation is split into two events:

  invocation  \[ A \; q\.\text{op}(a_1, ..., a_n) \] 
  response    \[ A \; q:\text{Ok}(r) \] 

• where \( A \) is a thread ID, \( q \) an object, \( \text{op}(a_1, ..., a_n) \) an invocation of call with arguments, and \( \text{Ok}(r) \) a successful response of call with result \( r \)

• a \textit{history} is a sequence of invocation / response events
Formal definitions

- A call of an operation is split into two events:
  - invocation \([A \ q.\text{op}(a_1, \ldots, a_n)]\)
  - response \([A \ q:\text{Ok}(r)]\)

- where \(A\) is a thread ID, \(q\) an object, \(\text{op}(a_1, \ldots, a_n)\) an invocation of call with arguments, and \(\text{Ok}(r)\) a successful response of call with result \(r\)

- A history is a sequence of invocation / response events

\[H = [A \ q.\text{enq}(2)], [B \ q.\text{enq}(5)], [B \ q.\text{Ok}], [A \ q.\text{Ok}], [B \ q.\text{deq}()], [B \ q.\text{Ok}(2)], [A \ q.\text{deq}()], [A \ q.\text{Ok}(5)]\]
Formal definitions

• we can define **projections** on objects and on threads

• assume we have a history:

\[ H = [A \ q1\ .\ enq(2) \ ], \ [B \ q2\ .\ enq(5) \ ], \ [B \ q2\ .\ Ok \ ], \ [A \ q1\ .\ Ok \ ], \ [B \ q1\ .\ deq() \ ], \ [B \ q1\ .\ Ok(2) \ ], \ [A \ q2\ .\ deq() \ ], \ [A \ q2\ .\ Ok(5) \ ] \]

• object projection:

\[ H|q1 = [A \ q1\ .\ enq(2) \ ], \ [A \ q1\ .\ Ok \ ], \ [B \ q1\ .\ deq() \ ], \ [B \ q1\ .\ Ok(2) \ ] \]

• thread projection:

\[ H|A = [A \ q1\ .\ enq(2) \ ], \ [A \ q1\ .\ Ok \ ], \ [A \ q2\ .\ deq() \ ], \ [A \ q2\ .\ Ok(5) \ ] \]
Formal definitions

- a response matches an invocation if their object and thread names agree

- a history is sequential if it starts with an invocation, and each invocation (except possibly the last) is immediately followed by a matching response

\[ H = [A \text{q.enq(2)}], [A \text{q.Ok}], [B \text{q.enq(5)}], [B \text{q.Ok}] \]

- a sequential history is legal if it agrees with the sequential specification of each object
Formal definitions

• a call op₁ precedes another call op₂ (op₁ -> op₂) if op₁’s response event occurs before op₂’s invocation event

• we write ->ₕ for the precedence relation induced by H

  => e.g. q.enq(2) ->ₕ q.enq(5)

• an invocation is pending if it has no matching response

• a history is complete if it does not have pending responses

• complete(H) is the subhistory of H with all pending invocations removed
Linearizability: the definition

• two histories $H$ and $G$ are equivalent if $H|A = G|A$ for all threads $A$

• a history $H$ is linearizable if it can be extended to a history $G$ by adding zero or more response events, such that:

  
  - $complete(G)$ is equivalent to some legal sequential history $S$
  - $\rightarrow_H \subseteq \rightarrow_S$ (i.e. the precedences of $H$ are maintained)
**Linearizability: the definition**

- two histories \( H \) and \( G \) are **equivalent** if \( H | A = G | A \) for all threads \( A \)

- a history \( H \) is **linearizable** if it can be extended to a history \( G \) by adding zero or more response events, such that:
  
  \[
  \Rightarrow \text{complete}(G) \text{ is equivalent to some legal sequential history } S
  \]
  
  \[
  \Rightarrow \rightarrow_H \subseteq \rightarrow_S \text{ (i.e. the precedences of } H \text{ are maintained)}
  \]

\[
\begin{align*}
  H & = \{a \rightarrow c, b \rightarrow c\} \\
  \rightarrow_H & \subseteq \rightarrow_S \\
  S & = \{a \rightarrow b, a \rightarrow c, b \rightarrow c\}
\end{align*}
\]
Next on the agenda

1. concurrent objects and correctness
2. quiescent and sequential consistency
3. linearizability
Final remarks

• correctness notions for concurrent objects boil down to “equivalences” with sequential computations

  => quiescent consistency, sequential consistency, linearizability
  => objects we built in previous weeks were linearizable

• correctness conditions depend on the application’s needs

• in most modern multiprocessor architectures, memory reads/writes are not sequentially consistent

  => too expensive!
  => must “ask” for it explicitly when needed
Lecture based on Chapter 3 of: 

*The Art of Multiprocessor Programming*

Maurice Herlihy & Nir Shavit

*recommended reading!*