What is a concurrent object?

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

- How do we describe one?
- How do we tell if we’re right?
Case study: FIFO queue

\[ q = \begin{array}{c}
2 \\
4 \\
\end{array} \]
Case study: FIFO queue

q = 2 4

q.enq(6)
Case study: FIFO queue

\[
q = \begin{array}{c}
2 \\
4 \\
6 \\
\end{array}
\]

q.enq(6)
Case study: FIFO queue

q = [2 4 6]

q.enq(6)
q.deq()
Case study: FIFO queue

q = 4 6

q.enq(6)
q.deq() ⇒ 2
Implementation: Lock-based ring buffer

```cpp
#include <array>

template <typename Element, int capacity>
class Lock_based_FIFO
{
public:
    void enq(Element);
    Element deq();

private:
    std::array<Element, capacity> data_;  
    unsigned head_ = 0, tail_ = 0;
    Lock lock_; 
};
```
template <typename Element, int capacity>
void Lock_based_FIFO<Element, capacity>::enq(Element x)
{
    LockGuard guard(lock_);
    if ((tail_ - head_ == capacity)) throw fifo_full();
    data_[tail_++ % capacity] = x;
}
Implementation: Lock-based dequeue

template<typename Element, int capacity>
Element Lock_based_FIFO<Element, capacity>::deq()
{
    LockGuard guard(lock_);
    if (tail_ == head_) throw fifo_empty();
    return data_[head_++ % capacity];
}
Now consider this

Same thing, but:

- no mutual exclusion
- only two threads:
  - one only enqueues
  - one only dequeues
Wait-free SRSW FIFO

```cpp
#include <array>
#include <atomic>

template <typename Element, int capacity>
class Wf_SRSW_FIFO
{
public:
    void enq(Element);
    Element deq();

private:
    std::array<Element, capacity> data_;  
    std::atomic<unsigned long> head_{0}, tail_{0};
};
```
Wait-free SRSW enqueue

template <typename Element, int capacity>
void Wf_SRSW_FIFO<Element, capacity>::enq(Element x)
{
    if (tail_ - head_ == capacity) throw fifo_full();
    data_[tail_ % capacity] = x;
    ++tail_;}

Wait-free SRSW deque

template <typename Element, int capacity>
Element Wf_SRSW_FIFO<Element, capacity>::deq()
{
    if (tail_ == head_) throw fifo_empty();

    Element result = data_[head_ % capacity];
    ++head_;
    return result;
}
What *is* a concurrent queue?

- Need a way to *specify* a concurrent queue object
- Need a way to *prove* that an algorithm implements the spec
What *is* a concurrent queue?

- Need a way to **specify** a concurrent queue object
- Need a way to **prove** that an algorithm implements the spec

How do we specify objects?
Object specification

In a concurrent setting:

- it gets the right answer (correctness, a safety property)
- it doesn’t get stuck (progress, a liveness property)

Let’s start with correctness.
Sequential objects

Each object has:

- a **state**:  
  - fields, usually  
  - FIFO example: the sequence of elements

- a set of **methods**:  
  - only way to access/manipulate the state  
  - FIFO example: `enq` and `deq` methods
Sequential specification

- If (precondition)
  - the object is in such-and-such a state
  - before you call the method,
Sequential specification

- 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
Sequential specification

- 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)
  - the object will be in some specified state
  - when the method returns.
Example sequential specification: dequeue

- **Precondition:**
  - queue state is $x_1, x_2, \ldots, x_k$ for $k \geq 1$

- **Postcondition:**
  - returns $x_1$

- **Postcondition:**
  - queue state is $x_2, \ldots, x_k$
Example sequential specification: dequeue

- **Precondition:**
  - queue state is $x_1, x_2, \ldots, x_k$ for $k \geq 1$

- **Postcondition:**
  - returns $x_1$

- **Postcondition:**
  - queue state is $x_2, \ldots, x_k$

Easy!
Example sequential specification: dequeue

- **Precondition:**
  - queue is empty

- **Postcondition:**
  - throws fifo_empty exception

- **Postcondition:**
  - state is unchanged

Easy!
Sequential specifications are awesome!

- All method interactions captured by side-effects on state
- Each method described in isolation
- Can add new methods easily
Sequential specifications are awesome!

- All method interactions captures by side-effects on state
- Each method described in isolation
- Can add new methods easily

What about concurrent specification?
Complication: methods take time

- Sequential: what is time? who cares?
- Concurrent: method call is interval, not event
Complication: methods take overlapping time

- Sequential: what is time? who cares?
- Concurrent: method call is interval, not event

- Sequential: invariants must hold between calls
- Concurrent: overlapping means might never be between calls
The Big Question

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

Or, what is a concurrent FIFO queue?
The Big Question

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

Or, what is a concurrent FIFO queue?

- FIFO means stuff happens in order
- concurrent means time/order is kinda ambiguous
Intuitively...

```cpp
template <typename Element, int capacity>
void Lock_based_FIFO<Element, capacity>::enq(Element x) {
    LockGuard guard(lock_);
    if (tail_ - head_ == capacity) throw fifo_full();
    data_[tail_++] % capacity = x;
}

template <typename Element, int capacity>
Element Lock_based_FIFO<Element, capacity>::deq() {
    LockGuard guard(lock_);
    if (tail_ == head_) throw fifo_empty();
    return data_[head_++] % capacity;
}
```
Intuitively...

template <typename Element, int capacity>
void Lock_based_FIFO<Element, capacity>::enq(Element x) {
    LockGuard guard(lock_);
    if (tail_ - head_ == capacity) throw fifo_full();
data_[tail_++] % capacity = x;
}

template <typename Element, int capacity>
Element Lock_based_FIFO<Element, capacity>::deq() {
    LockGuard guard(lock_);
    if (tail_ == head_) throw fifo_empty();
    return data_[head_++] % capacity;
}

Mutual exclusion means we can describe the behavior sequentially
Linearizability

- Each method “takes effect” “instantaneously” between invocation and response events
- Object is correct if this “sequential” behavior is correct
Linearizability

- Each method “takes effect” “instantaneously” between invocation and response events
- Object is correct if this “sequential” behavior is correct

Such a concurrent object is *linearizable*
Is linearizability really about the object?

A linearizable object: all of its possible executions are linearizable

(Linearizable execution examples on board)
Formal model of executions

Split method call into two events:

<table>
<thead>
<tr>
<th>Invocation</th>
<th>A q.enq(x)</th>
<th>Thread A calls q.enq(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response</td>
<td>A q:void</td>
<td>Result is void</td>
</tr>
</tbody>
</table>
Definition: History

\[ H = \]

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

Object projection:

\[ \frac{A \ q.\text{enq}(3)}{A \ q:\text{void}} \]

\[ H|q = \frac{B \ q.\text{deq}()}{B \ q:3} \]
Definition: History

Thread projection:

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

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

Remove pending invocations:

\[ H = \]

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

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

Remove pending invocations:

A q.enq(3)
A q: void

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

Responses immediately follow invocations (except possibly a final invocation):

\[ H = \]

\[
\begin{array}{l}
A \text{ q.enq}(3) \\
A \text{ q:} \text{void} \\
B \text{ p.enq}(4) \\
B \text{ p:} \text{void} \\
B \text{ q.deq}() \\
B \text{ q:} 3 \\
A \text{ q.deq}() \\
\end{array}
\]
History well-formedness

$H =$

A q.enq(3)
B p.enq(4)
B p:void
B q:deq()
B q:3
A q:void
History well-formedness

\[ H = \]

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

\( H \) is well formed if its thread projections are sequential:
History well-formedness

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

$H$ is well formed if its thread projections are sequential:

$H|A = \begin{align*}
\text{A q.enq(3)} \\
\text{A q: void} \\
\end{align*}$

$H|B = \begin{align*}
\text{B p.enq(4)} \\
\text{B p: void} \\
\text{B q.deq()} \\
\text{B q: 3} \\
\end{align*}$
History equivalence

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

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

iff threads see the same things:
History equivalence

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

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

\(G \sim H\) iff threads see the same things:

\[
H|A = G|A
\]

\[
H|B = G|B
\]
Sequential specification

A sequential specification describes a legal single-thread, single-object history
A sequential specification describes a legal single-thread, single-object history

A grammar for (unbounded) FIFO histories:

\[
H \quad ::= \quad H_\epsilon \\
H_{x_1, \ldots, x_k} \quad ::= \quad q.\text{enq}(x); \ q:\text{void}; \ H_{x_1, \ldots, x_k, x} \\
H_{x_0, x_1, \ldots, x_k} \quad ::= \quad q.\text{deq}(); \ q:x_0; \ H_{x_1, \ldots, x_k}
\]
Legal histories

A sequential (multi-object, multi-thread) history $H$ is legal if:

For every object $x$, $H|_x$ is in the sequential spec for $x$. 
**Precedence**

A method call $c$ *precedes* a method call $d$ if $c$’s response comes before $d$’s invocation.
**Precedence**

A method call \( c \) **precedes** a method call \( d \) if \( c \)'s response comes before \( d \)'s invocation.

**Example:**

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

- Method call \( A \text{ q.enq}(3) \) precedes method call \( B \text{ q.deq}() \)
- Method call \( A \text{ q.enq}(4) \) precedes method call \( B \text{ q.deq}() \)
- Method call \( A \text{ q.enq}(3) \) **does not precede** method call \( B \text{ q.enq}(4) \)
Properties of precedence

- In general, it’s a partial order
- For a sequential history, it’s a total order

Have we seen this before?
Properties of precedence

- In general, it’s a partial order
- For a sequential history, it’s a total order

Have we seen this before?

Yes: Precedence is *happens-before* (→) for method call intervals
Linearizability, formally

History $H$ is *linearizable* if it can be extended to complete history $G$ by

- appending responses to some pending invocations, and/or
- discarding the remaining pending invocations

such that there exists some legal sequential history $S \sim G$ where $\rightarrow_H \subseteq \rightarrow_S$
Example

\[ H = \]

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

\[ H = \]
- A q.enq(3)
- B q.enq(4)
- B q:void
- B q.deq()
- B q:4
- B q.enq(6)

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

\[ S \] is legal and sequential

34
Example

\[ H = \]
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q.enq(6)

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

\[ S = \]
B q.enq(4)
B q:void
A q.enq(3)
A q:void
B q.deq()
B q:4
### Example

<table>
<thead>
<tr>
<th>$H =$</th>
<th>$G =$</th>
<th>$S =$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A q.enq(3)</td>
<td>A q.enq(3)</td>
<td>B q.enq(4)</td>
</tr>
<tr>
<td>B q.enq(4)</td>
<td>B q.enq(4)</td>
<td>B q.enq(4)</td>
</tr>
<tr>
<td>B q:void</td>
<td>B q:void</td>
<td>B q:void</td>
</tr>
<tr>
<td>B q.deq()</td>
<td>B q.deq()</td>
<td>A q.enq(3)</td>
</tr>
<tr>
<td>B q:4</td>
<td>B q:4</td>
<td>A q:void</td>
</tr>
<tr>
<td>B q.enq(6)</td>
<td></td>
<td>B q.deq()</td>
</tr>
<tr>
<td>A q:void</td>
<td></td>
<td>B q:4</td>
</tr>
</tbody>
</table>

- **$S$** is legal and sequential
- **$S \sim G$**
- **$\rightarrow_H \subseteq \rightarrow_S$**
Composability theorem

History $H$ is linearizable if for every object $x$, $H|x$ is linearizable
Composability theorem

History $H$ is linearizable if for every object $x$, $H|_x$ is linearizable

This means we can reason about objects independently
This work is licensed under a Creative Commons “Attribution-ShareAlike 3.0 Unported” license.

These slides are derived from the companion slides for The Art of Multiprocessor Programming, by Maurice Herlihy and Nir Shavit. Its original license reads:

_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 of that work or this 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.