Concurrent Objects and Linearizability

EECS 395 “Rust”

Feb. 2, 2016

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 =	2	4		

Case study: FIFO queue

q = | 2 | 4 | | |

q.enq(6)

Case study: FIFO queue

q =	2	4	6	

q.enq(6)

Case study: FIFO queue

q =

2	4	6	

q.enq(6)

q.deq()

Case study: FIFO queue

q =

q.enq(6)

q.deq() ⇒ 2

Implementation: lock-based ring buffer

```
template ⟨typename Element, int capacity⟩
class Lock_based_FIFO
{
public:
    void enq(Element);
    Element deq();

private:
    array⟨Element, capacity⟩ data;
    unsigned head = 0, tail = 0;
    Lock lock;
};
```

Implementation: lock-based enqueue

```
template ⟨typename Element, int capacity⟩
void Lock_based_FIFO::enq(Element x)
{
    auto guard = lock.acquire();

    if (tail − head == capacity) throw fifo_full{};

    data[tail++ % capacity] = x;
}
```

Implementation: lock-based dequeue

```
template ⟨typename Element, int capacity⟩
Element Lock_based_FIFO::deq()
{
    auto guard = lock.acquire();

    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

```
template ⟨typename Element, int capacity⟩
class Wf_SRSW_FIFO
{
    array⟨Element, capacity⟩ data;
    unsigned head = 0, tail = 0;

public:
    void enq(Element) {
        if (tail − head == capacity) throw fifo_full{};
        data[tail++ % capacity] = x;
    }

    Element deq() {
        if (tail == head) throw fifo_empty{};
        return data[head++ % capacity];
    }
};
```

What *is* a concurrent queue?

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

What *is* a concurrent queue?

- Need a way 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 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 is non-empty
- Postcondition:
  - returns first item in queue
- Postcondition:
  - removes first item in queue

Example sequential specification: dequeue

- Precondition:
  - queue is non-empty
- Postcondition:
  - returns first item in queue
- Postcondition:
  - removes first item in queue

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

What about concurrent specifications?

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 even be between calls

The Big Question

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

The Big Question

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…

```
template ⟨typename Element, int capacity⟩
Element Lock_based_FIFO::deq() {
    auto guard = lock.acquire();
    if (tail == head) throw fifo_empty{};
    return data[head++ % capacity];
}

template ⟨typename Element, int capacity⟩
void Lock_based_FIFO::enq(Element x) {
    auto guard = lock.acquire();
    if (tail − head == capacity) throw fifo_full{};
    data[tail++ % capacity] = x;
}
```

Intuitively…

```
template ⟨typename Element, int capacity⟩
Element Lock_based_FIFO::deq() {
    auto guard = lock.acquire();
    if (tail == head) throw fifo_empty{};
    return data[head++ % capacity];
}

template ⟨typename Element, int capacity⟩
void Lock_based_FIFO::enq(Element x) {
    auto guard = lock.acquire();
    if (tail − head == capacity) throw fifo_full{};
    data[tail++ % capacity] = x;
}
```

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:

Invocation	A q.enq(x)	Thread A calls q.enq(x)
Response	A q:void	Result is void

History

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

History

Object projection:

$$H|\text{q} = \begin{matrix} \text{A q.enq(3)} \\ \text{A q:void} \\ \\ \text{B q.deq()} \\ \text{B q:3} \end{matrix}$$

History

Thread projection:

$H|\mathrm{B} =$

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

History

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

Complete subhistories

Remove pending invocations:

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

Complete subhistories

Remove pending invocations:

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

Sequential subhistories

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

$$H = \begin{array}{l} \text{A q.enq(3)} \\ \text{A q:void} \\ \hline \text{B p.enq(4)} \\ \text{B p:void} \\ \hline \text{B q.deq()} \\ \text{B q:3} \\ \hline \text{A q.enq(5)} \end{array}$$

History *well-formedness*

$H =$

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

History *well-formedness*

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

H is well formed if its thread projections are sequential:

History *well-formedness*

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

H is well formed if its thread projections are sequential:

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

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

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

History equivalence

$$H = \begin{array}{l} \text{A q.enq(3)} \\ \text{B p.enq(4)} \\ \text{B p:void} \\ \text{B q:deq()} \\ \text{A q:void} \\ \text{B q:3} \end{array} \qquad G = \begin{array}{l} \text{A q.enq(3)} \\ \text{A q:void} \\ \text{B p.enq(4)} \\ \text{B p:void} \\ \text{B q:deq()} \\ \text{B q:3} \end{array}$$

$G \sim H$ iff threads see the same things:

$$H|\text{A} = G|\text{A}$$
$$H|\text{B} = G|\text{B}$$

Sequential specifiction

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

Sequential specifiction

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

Example: CFG for sequential histories of unbounded FIFO:

$$
\begin{aligned}
\mathrm{BH} &::= \epsilon \\
\mathrm{BH} &::= \mathrm{BH};\ \mathrm{BH} \\
\mathrm{BH} &::= \mathrm{q.enq(x);\ q{:}void;\ BH;\ q.deq();\ q{:}x} \\
\mathrm{H} &::= \mathrm{BH} \\
\mathrm{H} &::= \mathrm{q.deq();\ q{:}Empty} \\
\mathrm{H} &::= \mathrm{H};\ \mathrm{H}
\end{aligned}
$$

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:

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

- Method call A q.enq(3) precedes method call B q.deq()
- Method call B q.enq(4) precedes method call B q.deq()
- Method call A q.enq(3) does not precede method call B 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* ($\rightarrow$) 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 where $\rightarrow_G \subseteq \rightarrow_S$

Example

$$H = \begin{array}{l} \text{A q.enq(3)} \\ \text{B q.enq(4)} \\ \text{B q:void} \\ \text{B q.deq()} \\ \text{B q:4} \\ \text{B q.enq(6)} \end{array}$$

S is legal and $\sim G$

Example

	A q.enq(3)		A q.enq(3)
	B q.enq(4)		B q.enq(4)
	B q:void		B q:void
$H =$	B q.deq()	$G =$	B q.deq()
	B q:4		B q:4
	B q.enq(6)		
			A q:void

S is legal and $\sim G$

Example

$$H = \begin{array}{l} \text{A q.enq(3)} \\ \text{B q.enq(4)} \\ \text{B q:void} \\ \text{B q.deq()} \\ \text{B q:4} \\ \text{B q.enq(6)} \end{array} \qquad G = \begin{array}{l} \text{A q.enq(3)} \\ \text{B q.enq(4)} \\ \text{B q:void} \\ \text{B q.deq()} \\ \text{B q:4} \\ \\ \text{A q:void} \end{array} \qquad S = \begin{array}{l} \text{B q.enq(4)} \\ \text{B q:void} \\ \text{A q.enq(3)} \\ \text{A q:void} \\ \text{B q.deq()} \\ \text{B q:4} \end{array}$$

S is legal and $\sim G$

Composability theorem

History *H* is linearizable iff for every object x, *H*|x is linearizable

Composability theorem

History *H* is linearizable iff for every object x, *H*|x is linearizable

This means we can reason about objects independently