Concurrent Objects

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit
Adapted by Larry Wittie for CSE391 S14
Concurrent Computation
Objectivism

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

• What is a concurrent object?
  – How do we describe one?

  – How do we tell if we’re right?
FIFO Queue: Enqueue Method

Head item in \( q \)

\[ q\text{.enq}(\circ) \]
FIFO Queue: Dequeue Method

\[ q.\text{deq}() \]
Lock-Based Queue

Lock glyphs show x y enq’d first

capacity = 8
Lock-Based Queue

Fields of q protected by a single shared lock

capacity = 8
Lock-Based Queue

After deq of x

Fields of q protected by a single shared lock

capacity = 8
A Lock-Based Queue

Code samples show $yz$ enq'd first

```java
class LockBasedQueue<T> {
    int head, tail;
    T[] items;
    Lock lock;
    public LockBasedQueue(int capacity) {
        head = 0; tail = 0;
        lock = new ReentrantLock();
        items = (T[]) new Object[capacity];
    }
}
```

Fields protected by a single shared lock
Lock-Based Queue

Initially head = tail
A Lock-Based Queue

class LockBasedQueue<T> {
    int head, tail;
    T[] items;
    Lock lock;

    public LockBasedQueue(int capacity) {
        head = 0; tail = 0;
        lock = new ReentrantLock();
        items = (T[]) new Object[capacity];
    }
}

Initially head = tail
Lock-Based `deq()`
Acquire Lock in \texttt{deq}()
Implementation: `deq()`

```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();
    }
}
```

Acquire lock at method start.
Check if Non-Empty

Waiting to enqueue…
Implementation: `deq()`

```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();
    }
}
```

If queue empty throw exception
Modify the Queue

Waiting to enqueue...
Implementation: `deq()`

```java
class Queue {
    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();
        }
    }
}
```

Queue not empty? Good!

Remove item and update head
Implementation: `deq()`

```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();
    }
}
```

Return result

Art of Multiprocessor Programming  Ch03
CSE391 Multicore Programming S14
2/11,18,20/14
At End of deq(), Release Lock

My turn!
Implementation: **deq()**

```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();
    }
}
```

Release lock no matter what!
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();
    }
}
Now consider the following implementation

- The same thing without mutual exclusion
- For simplicity, only two threads
  - One thread enq only
  - The other deq only
Wait-free 2-Thread Queue

capacity = 8
Wait-free 2-Thread Queue

deq()

enq(z)

head

tail
Wait-free 2-Thread Queue

result = x

queue[tail] = z
Wait-free 2-Thread Queue

head++

tail--

head

tail

x

y

z

0 1 2 3 4 5 6 7
public class WaitFreeQueue {

    int head = 0, tail = 0;
    items = (T[]) new Object[capacity];

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

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

No lock needed!
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();
    }
}
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 …
Correctness and Progress

• In a concurrent setting, we need to specify both the safety and the liveness properties of an object

• Need a way to define
  – when an implementation is correct
  – the conditions under which it guarantees progress

Let’s begin with correctness
Sequential Objects

• Each object has a *state* (memory values)
  – 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
  – via “side-effects” when the method returns.
Pre- and Post-Conditions for Dequeue

• **Precondition:**
  – Queue is non-empty

• **Postcondition:**
  – Returns first item in queue

• **Postcondition:**
  – Removes first item in queue
Pre- and Post-Conditions for Dequeue

• **Precondition:**
  – Queue is empty

• **Postcondition:**
  – Throws Empty exception

• **Postcondition:**
  – Queue state unchanged
Why Sequential Specifications Totally Rock = Are So Simple

- 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
What About Concurrent Specifications?

• Methods?
• Documentation?
• Adding new methods?
Methods Take Time
Methods Take Time

invocation 12:00

q.enq( )
Methods Take Time

invocation
12:00

q.enq(0)

Method call

time
Methods Take Time

Invocation 12:00

Method call

time
Methods Take Time

invocation 12:00

q.enq(...) -> Method call

response 12:01

void
Sequential vs Concurrent Objects

- **Sequential**
  - Methods take time? Who knew?

- **Concurrent**
  - Method call is not an event
  - Method call is an interval.
Concurrent Methods Take Overlapping Time
Concurrent Methods Take Overlapping Time
Concurrent Methods Take Overlapping Time
Concurrent Methods Take Overlapping Time
Sequential vs Concurrent

• Sequential:
  – Object needs meaningful state only \textit{between} method calls

• Concurrent
  – Because method calls overlap, object might \textit{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`s overlap?
    • Two `deq`s? `enq` and `deq`? …
Sequential vs Concurrent

• **Sequential:**
  – Can add new methods without affecting older methods

• **Concurrent:**
  – Everything can potentially interact with everything else
Sequential vs Concurrent

• **Sequential:**
  – Can add new methods without affecting older methods

• **Concurrent:**
  – 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 ... \textbf{Locks Simplify}

```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();
    }
}
```
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
Intuitively

Let’s capture the idea of describing the concurrent via the sequential executions.

Locks cut concurrent intervals into separate sequential executions.

Behavior is “Sequential”
Linearizability

- Each method should
  - “take effect”
  - Instantaneously
  - Between invocation and response events

- Object is correct if this “sequential” behavior is correct

- Any such concurrent object is
  - Linearizable™
Is it really about the object?

- Each method should
  - “take effect”
  - Instantaneously
  - Between invocation and response events
- Sounds like a property of an execution…
- A linearizable object: one all of whose possible executions are linearizable
Example
Example

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

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

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

\[ \text{time} \]
Example

\[ q\.enq(x) \]
\[ q\.enq(y) \]
\[ q\.deq(x) \]

\textbf{time}
Example

Thread A

Thread B

q.enq(x)

q.enq(y)

q.deq(x)

q.deq(y)

time
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

Thread A

Thread B

A

q.enq(x)

q.enq(y)

B

q.enq(y)

q.deq(x)

q.deq(y)

Valid?

time
Example
Example

q.enq(x)

time
Example

```
q.enq(x)
```

```
q.deq(y)
```

(time)
Example

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

\[ \text{time} \]
Example

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

B
time
Example

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

(5)

not linearizable
Example

\[ q.enq(x) \]
\[ q.enq(y) \]
\[ q.deq(y) \]
\[ q.enq(x) \]
\[ q.enq(y) \]

must be
\[ q.deq(x) \]

not linearizable
Example

\[ \text{time} \]
Example

Half-arrow means invoked method has not yet responded.

$q.enq(x)$

Time
Example

A

q.enq(x)

B

q.deq(x)

time
Example

Art of Multiprocessor Programming  Ch03
CSE391 Multicore Programming S14
2/11,18,20/14

\( q.{\text{enq}}(x) \)

\( q.{\text{deq}}(x) \)

time
Example

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

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

linearizable

A

B

time
Example

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

time
Example

q.enq(x)

q.enq(y)

time
Example

\[ q_{\text{enq}}(x) \]
\[ q_{\text{enq}}(y) \]
\[ q_{\text{deq}}(y) \]
Example

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

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

linearizable
Example

Comme ci

Comme ça

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

A

B

time

multiple orders OK

linearizable
Read/Write Register Example

\[ \text{write}(0) \quad \text{read}(1) \quad \text{write}(2) \quad \text{write}(1) \quad \text{read}(0) \]

\text{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(1) already happened

write(1)

write(2)

read(0)
Read/Write Register Example

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

write(1) already happened

not linearizable
Read/Write Register Example

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

write(1) already happened

must read(1 or 2)

not linearizable

read(0)
Read/Write Register Example

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

write(1) already happened
Read/Write Register Example

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

write(1) already happened
Read/Write Register Example

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

write(1) already happened

must read(2)

not linearizable
Read/Write Register Example

With no read(1)

write(0) write(2)

write(1) read(1)

time
Read/Write Register Example

With no `read(1)`, the `write(1)` can be later.

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

time
Read/Write Register Example

With no read(1), the write(1) can be later.
Read/Write Register Example

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

time
Read/Write Register Example

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

time
Read/Write Register Example

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

time

Art of Multiprocessor Programming  Ch03
CSE391 Multicore Programming S14
2/11,18,20/14
Read/Write Register Example

must read(2)

Not linearizable

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

write(1)

time

Art of Multiprocessor Programming  Ch03
CSE391 Multicore Programming S14
2/11,18,20/14
Read/Write Register Example

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

write(0) → read(1) → write(1) → write(2) → read(2)
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
Formal Model of Executions

• 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!
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)
Invocation Notation

```
A q.enq(x)
```
Invocation Notation

$q\text{.enq}(x)$

thread method
Invocation Notation

A q.enq(x)

thread  method  object
Invocation Notation

A q.enq(x)

thread  method  object  arguments

(4)
Response Notation

A q: void
Response Notation

A q: void

thread
Response Notation

thread result

A q: void
Response Notation

A q: void

thread

result

object
Response Notation

Method is implicit

thread

object

result

A q: void

(2)
Response Notation

A q: empty()

Method is implicit

thread

exception

object
History - Describing an Execution

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

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

\[ \text{A q: void} \]
Object Projections

H =

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

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

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

Subhistory H at object q (no p’s) “H at q”
Thread Projections

\[ H = \]

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

\[ H|B = \]

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

Subhistory H at thread B (no A's) “H at B”
Complete Subhistory

An invocation is **pending** if it has no matching response.
Complete Subhistory

H =

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

May or may not have taken effect
Complete Subhistory

\[ H = \]

A q.enq(3)
A q: void
A q.enq(5)

B p.enq(4)
B p: void
B q.deq()
B q: 3
discard pending invocations
Complete Subhistory

A \texttt{q.enq(3)}
A \texttt{q: void}

\textbf{Complete}(H) =

B \texttt{p.enq(4)}
B \texttt{p: void}
B \texttt{q.deq()}
B \texttt{q: 3}
Sequential Histories

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

A q.enq(3)
A q:void

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

match
Sequential Histories

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

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

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

B q.deq()  match
B q: 3

A q: enq(5)
Sequential Histories

A q.enq(3)
A q: void

B p.enq(4)
B p: void

B q.deq()
B q: 3

A q: enq(5)

match
match
match
Final pending invocation OK
Sequential Histories

A q.enq(3)
A q: void

B p.enq(4)
B p: void

B q.deq()
B q: 3

Method calls of different threads do not interleave

match

match

Final pending invocation OK

(4)
Well-Formed Histories

\[ H = \]

1. A $q.enq(3)$
2. B $p.enq(4)$
3. B $p: void$
4. B $q.deq()$
5. A $q: void$
6. B $q: 3$
Well-Formed Histories

Per-thread projections sequential

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

\[ H | B = \]
B p.enq(4)
B p:void
B q.deq()
B q:3
Well-Formed Histories

Per-thread projections sequential

\[ H = \]

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

\[ H | B = \]

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

\[ H | A = \]

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

Threads see the same projections in both

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

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

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

G is sequential
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 $x$
  – $H|x$ is in the sequential specification for $x$
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 the first’s response event precedes the invocation event of the second.
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

Method call
Method call
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 in general
  – Total order if $H$ is sequential
Note on Linearizability

• In general, given a history \( H \), precedence only defines a partial order
  – \( H \) can have methods that overlap
• For a sequential history \( S \), precedence defines a total order, because
  – \( S \) has no overlapping methods
• Ambiguity about precedence order can be fixed by linearizability if \( H \)'s partial order is somehow completed to a total order
• The next slide defines linearizability of \( H \)
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 $S$
    • \{Legal: for all objects $x$ in $S$, $S|x$ is sequential\}
  – Where $\rightarrow^*_G \subseteq \rightarrow^*_S$
Extending a Linearizable History

- Extend history $H$ to $G$ with no pending calls
  - Append response(s) or discard invocation(s)
- Find a sequential history $S$ equivalent to $G$
  - Where, for all objects $x$ in $S$, $S|x$ is sequential
  - And sequential $S$‘s total order includes all the method precedences of the partial order in $G$
  - All $S|x$ are sequential so responses are correct
  - $G$ is equivalent to $S$, so $G$‘s method invocation orders & results are right for each object $x$ in $G$
- If $G$ and $S$ are found for $H$, $H$ is linearizable
Ensuring $\Rightarrow_G \subseteq \Rightarrow_S$

$\Rightarrow_G = \{a \Rightarrow c, b \Rightarrow c\}$

$\Rightarrow_S = \{a \Rightarrow b, a \Rightarrow c, b \Rightarrow c\}$

A limitation on the Choice of $S$!
Remarks

- Some pending invocations
  - Took effect, so keep them
  - Discard the rest

- Condition $\rightarrow G \subset \rightarrow S$
  - 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)

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

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

time
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 one
Example

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

discard this one

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

time
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
```

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

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

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

- **Focus on total methods**
  - Response defined in every state of object

- **Example:**
  - `deq()` that throws `Empty` exception
  - Versus `deq()` that waits …

- **Why?**
  - Otherwise, blocking unrelated to synchronization
Concurrent Programming

• **Question:** When does linearizability require a method invocation to block?
• **Answer:** never.
• Linearizable total methods are *non-blocking*
Non-Blocking if Linearizable Theorem

If method invocation

\[ A \ q.\text{inv}(...) \]

is pending in history \( H \), then there exists a response to append to \( H \)

\[ A \ q.\text{res}(...) \]

such that

\[ H + A \ q.\text{res}(...) \]

is linearizable
Proof

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

• History $H$ is linearizable if and only if
  – For every object $x$
  – $H|_x$ is linearizable
• We care about objects only!
  – (Materialism?)
Why Does Composability Matter?

• Modularity
• Can prove linearizability of objects in isolation
• Can compose independently-implemented objects
Reasoning About Linearizability: Locking

```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();
    }
}
```
Reasoning About Linearizability: Locking

```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();
    }
}
```

Linearization points are when locks are released.
More Reasoning: Wait-free

```
public class WaitFreeQueue {

    int head = 0, tail = 0;
    items = (T[]) new Object[capacity];

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

    public Item deq() {
        if (tail == head) throw new EmptyException();
        Item item = items[head % capacity]; head++;
        return item;
    }
}
```
More Reasoning: Wait-free

Remember that there is only one enqueuer and only one dequeuer. Linearization order is order head and tail fields are modified.

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

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

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

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

• Doesn’t always work
  – Might need to define several different steps for a given method (see Lists chapter)
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 $S$

  –Where $G \subset S$

  Differs from linearizability
Sequential Consistency

• No need to preserve real-time order
  – Cannot re-order operations done by the same thread
  – Can re-order non-overlapping operations done by different threads

• Often used to describe multiprocessor memory architectures
Example
Example

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

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

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

Time
Example

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

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

not linearizable
Example

Yet Sequentially Consistent

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

Why Sequentially Consistent

\[
\begin{align*}
q.enq(x) & \quad q.enq(y) \\
q.deq(y) & \quad q.enq(x) \\
q.enq(y) & \quad q.enq(y)
\end{align*}
\]
Theorem

Sequential Consistency is not composable
FIFO Queue Example

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

`time`
FIFO Queue Example

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

`time`
FIFO Queue Example

History H

Thread A

Thread B

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

\[ \text{time} \]
H\(\parallel\)p Sequentially Consistent

A

\[ p.enq(x) \]

\[ q.enq(x) \]

\[ p.deq(y) \]

\[ q.enq(y) \]

\[ p.enq(y) \]

\[ q.deq(x) \]

B

time

Art of Multiprocessor Programming  Ch03
CSE391 Multicore Programming S14
2/11,18,20/14
H|q Sequentially Consistent

(time)

A

p.enq(x) → q.enq(x) → p.deq(y)

B

q.enq(y) → p.enq(y) → q.deq(x)

time
Ordering imposed by p

A

p.enq(x)

q.enq(x)

p.deq(y)

q.deq(x)

B

p.enq(y)

q.enq(y)

p.enq(y)

time
Ordering imposed by q

A

B

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

time
Ordering imposed by both

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

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

A

B

\text{time}
Combining orders

Cycle means there is no sequentially consistent order
Fact

• Most hardware architectures don’t support sequential consistency
• Because they think it’s too strong
• Here’s another story …
The Flag Example

\[\text{x.write}(1)\]

\[\text{y.write}(1)\]

\[\text{y.read}(0)\]

\[\text{x.read}(0)\]
The Flag Example

- Each thread’s view is sequentially consistent
  - It went first

\[ x\text{.write}(1) \rightarrow y\text{.write}(1) \rightarrow y\text{.read}(0) \rightarrow x\text{.read}(0) \]
The Flag Example

- Entire history isn’t sequentially consistent
  - Can’t both go first
The Flag Example

• Is this behavior really so wrong?
  – We can argue either way …
Opinion 1: It’s Wrong

• This pattern
  – Write mine, read yours
• Is exactly the flag principle
  – Beloved of Alice and Bob
  – Heart of mutual exclusion
    • Peterson
    • Bakery, etc.
• It’s non-negotiable!
Opinion2: But It Feels So Right …

- Many hardware architects think that sequential consistency is too strong
- Too expensive to implement in modern hardware
- OK if flag principle
  - violated by default
  - Honored by explicit request
Memory Hierarchy

• On modern multiprocessors, processors do not read and write directly to memory.
• Memory accesses are very slow compared to processor speeds,
• Instead, each processor reads and writes directly to a cache
Memory Operations

• **To read a memory location,**
  – load data into cache.

• **To write a memory location**
  – update cached copy,
  – lazily write cached data back to memory
While Writing to Memory

• A processor can execute hundreds, or even thousands of instructions
• Why delay on every memory write?
• Instead, write back in parallel with rest of the program.
Revisionist History

- Flag violation history is actually OK
  - processors delay writing to memory
  - until after reads have been issued.
- Otherwise unacceptable delay between read and write instructions.
- Who knew you wanted to synchronize?
Who knew you wanted to synchronize?

- Writing to memory = mailing a letter
- Vast majority of reads & writes
  - Not for synchronization
  - No need to idle waiting for post office
- If you want to synchronize
  - Announce it explicitly
  - Pay for it only when you need it
Explicit Synchronization

• **Memory barrier instruction**
  – Flush unwritten caches
  – Bring caches up to date
• Compilers often do this for you
  – Entering and leaving critical sections
• Expensive
Volatile

- In Java, can ask compiler to keep a variable up-to-date with volatile keyword
- Also inhibits reordering, removing from loops, & other “optimizations”
Real-World Hardware Memory

• Weaker than sequential consistency
• But you can get sequential consistency at a price
• OK for expert, tricky stuff
  – assembly language, device drivers, etc.
• Linearizability more appropriate for high-level software
Linearizability

• Linearizability
  – Operation takes effect instantaneously between invocation and response
  – Uses sequential specification, locality implies composability
  – Good for high level objects
Correctness: Linearizability

• Sequential Consistency
  – Not composable
  – Harder to work with
  – Good way to think about hardware models

• We will use *linearizability* as in the remainder of this course unless stated otherwise
Progress

• We saw an implementation whose methods were lock-based (deadlock-free)
• We saw an implementation whose methods did not use locks (lock-free)
• How do they relate?
Progress Conditions

- **Deadlock-free**: some thread trying to acquire the lock eventually succeeds.
- **Starvation-free**: every thread trying to acquire the lock eventually succeeds.
- **Lock-free**: some thread calling a method eventually returns.
- **Wait-free**: every thread calling a method eventually returns.
## Progress Conditions

<table>
<thead>
<tr>
<th></th>
<th>Non-Blocking</th>
<th>Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Everyone</strong></td>
<td>Wait-free</td>
<td>Starvation-free</td>
</tr>
<tr>
<td>makes progress</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Someone</strong></td>
<td>Lock-free</td>
<td>Deadlock-free</td>
</tr>
<tr>
<td>makes progress</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary

• We will look at *linearizable blocking* and *non-blocking* implementations of objects.
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/](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.