Concurrent Objects

Adapted from the Companion slides for The Art of Multiprocessor Programming by Maurice Herlihy & Nir Shavit
Concurrent Computation

memory

object

object
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

$q.\text{enq} (\bigcirc)$
FIFO Queue: Dequeue Method

$q$.deq() / 0
Lock-Based Queue

capacity = 8
Lock-Based Queue

Fields protected by single shared lock

capacity = 8
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];
    }
}

Fields protected by 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

My turn...

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();
    }
}
```
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
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?
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
Release the Lock

My turn!
Implementation: `deq()`

```java
class ArrayQueue {
    private T[] items;
    private int head, tail;
    private int capacity;

    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 (no lock)
- 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

- head
- tail
- deq()
- enq(z)
Wait-free 2-Thread Queue

result = x

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

head++

head

0

1

head++

0

7

2

5

4

3

6

x

y

z

tail

tail--
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

Wait-free 2-Thread Queue
Wait-free 2-Thread Queue

- From Amdahl’s law: concurrent object where methods hold lock exclusively are less desirable than ones with finer-grained locking or no lock at all.
public T deq() throws EmptyException {
    lock.lock();
    try {
        if (tail == head)
            throw new EmptyException();
        T x = items[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
• Lets 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 (Correctness and Progress)
- Need a way to define
  - when an implementation is correct
  - the conditions under which it guarantees progress

Lets begin with correctness
Sequential Objects

• Each object has a **state**
  – Usually given by a set of **fields**
  – An object has well defined state
  – Queue example: sequence of items

• Each object has a set of **methods**
  – Only way to manipulate state
  – Queue example: `enq` and `deq` methods
Sequential Objects

• To describe object method: from plain English to formal specs.
• Application Programming Interfaces (APIs)
  – preconditions
  – inputs
  – postconditions
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 (side effect)
  – when the method returns,
Pre and PostConditions for Dequeue

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

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

• **Postcondition:**
  – Removes first item in queue (side effect)
Pre and PostConditions for Dequeue

• **Precondition:**
  – Queue is empty

• **Postcondition:**
  – Throws Empty exception

• **Postcondition:**
  – Queue state unchanged

• **This style of documentation is called a sequential specification.**
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
What About Concurrent Specifications?

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

-time-
Methods Take Time

invocation 12:00

q.enq( )

time
Methods Take Time

```
q.enq()
```

Invocation: 12:00
Methods Take Time

Invocation: 12:00

Method call

time
Methods Take Time

invocation 12:00

q.enq(...)

Method call

response 12:01

void

time
Sequential vs Concurrent

- **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

Method call

time

Art of Multiprocessor Programming
Concurrent Methods Take Overlapping Time

Method call

Method call
Concurrent Methods Take Overlapping Time

Method call

Method call

Method call

time
Sequential vs Concurrent

- **Sequential:**
  - Object needs meaningful state only *between* method calls

- **Concurrent**
  - Because method calls overlap, object might *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
  – Method calls can encounter object state with incomplete effects of other method calls
Sequential vs Concurrent

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

• Concurrent:
  – Everything can potentially interact with everything else
  – Method calls can encounter object state with incomplete effects of other method calls

Panic!
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();
    }
}
```
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.

Behavior is “Sequential”
Quiescent Consistency

• A method call is an **interval** started by an **invocation** and ending with a **response**

• A method is **pending** if it has been invoked but has not yet responded
Quiescent Consistency

Thread A

\[ r.\text{write}(7) \]

Thread B

\[ r.\text{write}(-3) \quad r.\text{read}(-7) \]

Wrong!
Quiescent Consistency

Principle 3.3.1
An object is quiescent if it has no pending method calls (Method call should appear to happen in a one-at-a-time sequential order).

Principle 3.3.2
Methods separated by a period of quiescence should appear to take effect in their real-time order.
Quiescent Consistency

Example: Queue

- Suppose x and y are concurrently enqueued.
- The queue becomes quiescent.
- z is enqueued.

- We may not know the order of x and y but they are before z.
Quiescent Consistency

• Quiescent consistency is a correctness property.
• Informally, any time an object becomes quiescent, then the execution so far is equivalent to some sequential execution of the completed call.
Quiescent Consistency

Example: Quiescent consistent shared counter

• Return number although may not be in the order of invoking getAndIncrement() but never duplicates
Quiescent Consistency

• A method is total if it is defined for every object state; otherwise it is partial.
  E.g. FIFO `enq()` is total but `deq()` is partial.

• In any concurrent exec, for pending invocation of a total method, there is a quiescent consistent response. *(non-blocking correctness condition)*
Quiescent Consistency

• A correctness property $P$ is **compositional** if whenever each object in a system satisfies $P$, the whole system also satisfies $P$.

• Components are proved correctly independently and result of composing the interfaces that rely on one another should be correct.
Quiescent Consistency

- Quiescent consistency is compositional
- Quiescent consistent objects can be composed to construct more complex ones.
Sequential Consistency

- The order in which a single thread issues method calls is called its program order.
Sequential Consistency

**Principle 3.3.1**
An object is quiescent if it has no pending method calls (Method call should appear to happen in a one-at-a-time sequential order).

**Principle 3.4.1** - Method calls should appear to take effect in program order.

**Principles 3.3.1 and 3.4.1** define sequential consistency.
Sequential Consistency

• Sequential consistency requires method calls to be ordered sequentially in a way consistent with program order.

• Method calls in a concurrent exec can be ordered sequentially such that:
  – the sequence is consistent with program order, and
  – the sequence meets the object’s sequential specification.
Sequential Consistency

Example
A enqueues x while B enqueues y; A dequeues y while B dequeues x

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

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

- A enq(x); B enq(y); B deq(x); A deq(y)
- B enq(y); A enq(x); A deq(y); B deq(x)

Both are consistent with method call’s program order – execution is sequentially consistent.
Sequential Consistency and Quiescent Consistency

- Sequential consistency and quiescent consistency are not comparable.
- Quiescent consistency may not preserve program order and sequential consistent is not affected by quiescent periods.
Sequential Consistency

Example

• A enqueues x, b enqueues y then A dequeues y.

- Is this sequentially consistent?
Yes

May violate FIFO queue but enq(x) and enq(y) are unrelated by program order because they are called by separate threads. Sequential consistency can reorder them.
Sequential Consistency

• Sequential consistency is non-blocking; any pending call to a total method can always be completed.

• Sequential consistency is **not** compositional: composing multiple sequentially consistent object may not be sequentially consistent.
Sequential Consistency - Not Compositional

- $p \text{ enq}(x)$
- $q \text{ enq}(x)$
- $p \text{ deq}(y)$

- $q \text{ enq}(y)$
- $p \text{ enq}(y)$
- $q \text{ deq}(x)$

- $p$ and $q$ are sequentially consistent but the whole execution is not.
- Why?
• Assume the method calls can be reordered to form a correct FIFO queue.
• Because A dequeues y from p, 
  \((p.\text{enq}(y) B) \rightarrow (p.\text{enq}(x) A)\)
• Because B dequeues x from q, 
  \((q.\text{enq}(x) A) \rightarrow (q.\text{enq}(y) B)\)
• Program order implies that 
  \((p.\text{enq}(x) A) \rightarrow (q.\text{enq}(x) A)\)
Linearizability

- Sequential consistency is not compositional!
- 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
Linearization Point

- Every linearizable execution is sequentially consistent but not vice versa.
- To know, identify a linearization for each method where the method takes effect.
- Lock implementation: method’s CS
- Non-lock: single step where the effects of the method become visible to other method calls.
Linearization Point

- Example: FIFO Queue
- What are the possible linearization points?

```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 Point

• Example: FIFO Queue
• What are the possible linearization points?
  • For deq(), when the head field is updated or when it throws EmptyException.
  • For enq(), when the tail field is updated.
Example
Example

\texttt{q.enq(x)}

\textbf{time}
Example
Example

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

\[ \text{time} \]
Example

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

Art of Multiprocessor Programming
Example

```plaintext
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

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

Valid?
Example

\[ q.enq(x) \]
Example

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

time
Example

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

q.enq(y)

time
Example

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

Time progression:

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

Art of Multiprocessor Programming
Example

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

(5)

not linearizable
Example
Example

q.enq(x)

time
Example

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

```
q.deq(x)
```

`time`
Example

```
qu.enqueue(x)
qu.dequeue(x)
qu.enqueue(x)
qu.dequeue(x)
```
Example

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

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

\[ \text{time} \]

linearizable
Example

\texttt{q.enq(x)}
Example

time

q.enq(x)

q.enq(y)
Example

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

q.enq(y)
Example

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

Art of Multiprocessor Programming

Example

Comme ci
Comme ça

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

thread
Invocation Notation

A thread

method

q.enq(x)
Invocation Notation

Object

Method

Thread

A q.enq(x)
Invocation Notation

A q.enq(x)

thread method object arguments
Response Notation

A q: void
Response Notation

A q: void

thread
Response Notation

```
A q: void
```

thread

result
Response Notation

A q: void

thread

result

object
Response Notation

Method is implicit

A q: void

thread

result

object
Response Notation

Method is implicit

Thread

Object

A q: empty()
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*}

History is a finite sequence of invocations and responses
Definition

• Invocation & response *match* if

Thread names agree

Object names agree

A q.enq(3)
A q: void

Method call
Object Projections

\[ H = \]

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

For object q, object subhistory is the sequence of all events in H with object q.
Object Projections

\[ H|q = \]

A \ q,enq(3)
A \ q,void
B \ q,void
B \ q,deq()
B \ q:3

For object q, object subhistory is the sequence of all events in H with object q.
Thread Projections

\[ H = \]

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

For thread B, thread subhistory is the sequence of all events in H with thread B
Thread Projections

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

For thread B, thread subhistory is the sequence of all events in H with thread B
An invocation is *pending* if it has no matching response.
Complete Subhistory

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

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 q.enq(3)
A q:void

Complete(H) =
B p.enq(4)
B p:void
B q.deq()
B 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)

A history is sequential if the first event is an invocation and each invocation has a matching response, except possibly the last.
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 \texttt{q.enq}(3)
A \texttt{q: void}

B \texttt{p.enq}(4)
B \texttt{p: void}
B \texttt{q.deq}()
B \texttt{q: 3}
A \texttt{q.enq}(5)

match
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)

match

match

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)

match
match
match

Final pending invocation OK
Sequential Histories

Method calls of different threads do not interleave.

Final pending invocation OK.
Well-Formed Histories

A history $H$ is well formed if each thread subhistory is sequential.

Note: object subhistory may not be sequential.
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

Art of Multiprocessor Programming
Well-Formed Histories

Per-thread projections sequential

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

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

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

Threads see the same thing in both for every thread

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

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

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

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

Method call \hspace{1cm} Method call
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
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
  – Total order if $H$ is sequential
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 $S$
  - where $\rightarrow_G \subset \rightarrow_S$
Ensuring $\rightarrow_G \subset \rightarrow_S$

$\rightarrow_G = \{a \rightarrow c, b \rightarrow c\}$
$\rightarrow_S = \{a \rightarrow b, a \rightarrow c, b \rightarrow c\}$

A limitation on the Choice of $S$!
Remarks

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

• Condition $G \subset S$
  – 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

Art of Multiprocessor Programming
Example

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

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

Equivalent sequential history

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

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

• Modularity
• Can prove linearizability of objects in isolation
• Can compose independently-implemented objects
Concurrency

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

• Focus on *total* methods
  – Defined in every state
• Example:
  – `deq()` that throws `Empty` exception
  – Versus `deq()` that waits …
• Why?
  – Otherwise, blocking unrelated to synchronization
Concurrency

• **Question**: When does linearizability require a method invocation to block?

• **Answer**: never.

• **Linearizability is** *non-blocking*
Non-Blocking Theorem

If method invocation

\[ A \ q.\ inv(\ldots) \]

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

\[ A \ q:\ res(\ldots) \]

such that

\[ H + A \ q:\ res(\ldots) \]

is linearizable
Proof

- Pick linearization $S$ of $H$
- If $S$ already contains
  - Invocation $A \ q.inv(...) \text{ and response}$
  - Then we are done.
- Otherwise, pick a response such that
  - $S + A \ q.inv(...) + A \ q.res(...)$
  - Possible because object is total.
Linearizability: Summary

- Powerful specification tool for shared objects
- Allows us to capture the notion of objects being “atomic”
Linearizability: Summary

• What do we use it for?
  – Non-blocking implementations.

• How do we use it?
  – Identify linearization points.
  – Construct sequential execution histories.
  – Ensure that objects are in a consistent state.
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

(time: 5)
Example

q.enq(x)

(time)
Example

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

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

(time)
Example

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

q.enq(y)

time
Example

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

(5)
Example

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

(time)

Not linearizable
Example

Yet Sequentially Consistent

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

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

(time)

(5)
Art of Multiprocessor Programming
Fact

• Most hardware architectures don’t support sequential consistency
• Because they think it’s too strong
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

Refer to lock-based queue in Figure 3.1:

• Suppose the queue is initially empty;
• A halts midway during its call to enq();
• B calls deq();
• B's call to deq() is nonblocking (has a response) but must acquire a lock;
• Subsequently, this becomes a blocking implementation.
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();
    }
}
Progress Conditions

• An implementation is blocking if an unexpected delay by one thread prevent the others from making progress.
• Unexpected delays are common in multiprocessors
• Severity depends on type of delay (cache miss, page fault, preemption)
Progress Conditions

• A method is wait-free if it guarantees that every call finishes its execution in a finite number of steps.

• Bounded wait-free if there is a bound on the number of steps. e.g. Bakery Algorithms doorway section (bound is the number of threads)
Bakery Algorithm

class Bakery implements Lock {
    ...
    public void lock() {
        flag[i] = true;
        label[i] = max(label[0], ..., label[n-1])+1;
        while (∃k flag[k]
            && (label[i],i) > (label[k],k));
    }
}
Progress Conditions

- An object is wait-free if its methods are wait-free.
- A class is wait-free if all instances of its objects are wait-free.
- Wait-free is a non-blocking progress condition; that is, an arbitrary and unexpected delay by one thread does not necessarily prevent the others from making progress.
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;
    }
}
Progress Conditions

• **Wait-free advantage:**
  – Guarantees that every thread makes progress.

• **Wait-free disadvantages:**
  – Algorithms are more complex, and
  – less efficient.
Progress Conditions

• A method is lock-free if it guarantees that infinitely often some method call finishes in a finite number of steps.
• Wait-free method implementation is also lock-free but not vice-versa
• Practically, a fast-lock free may be better than a slow wait-free algorithm where starvation is unlikely.
Dependent Progress Conditions

• *Wait-free* and *lock-free* are independent of the underlying platform;
• *Deadlock-free* and *starvation-free* are not:
  – Depend on if OS guarantees that every thread eventually leaves critical sections.
• *Dependent progress condition*: progress occurs only if the underlying platform provides certain guarantees.
Dependent Progress Conditions

• A method call executes in *isolation* if no other threads take any steps while it executes.

• A method is *obstruction-free* if, from any point after which it executes in isolation, it finishes in a finite number of steps.
Progress Conditions Summary

• *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.
Choosing Progress Conditions

- Needs of the application
- Characteristics of the underlying platform
- Absolute wait-free and lock-free progress properties:
  - Real-time;
- Dependent obstruction-free, deadlock-free and starvation free:
  - Guarantees by platform, simple and efficient implementations.
Java Memory Model

• Java does not guarantee linearizability or sequential consistency when reading or writing fields of shared objects.

• Why?
Java Memory Model

Java makes use of a relaxed memory model, which satisfies a fundamental property: if a program’s sequentially consistent executions follow certain rules then all executions of that program in the relaxed memory model will be sequentially consistent.

What does this mean?
public static Singleton getInstance() {
    if (instance == null) {
        synchronized(Singleton.class) {
            if (instance == null)
                instance = new Singleton();
        }
    }
    return instance;
}
Shared vs. Local Memory

```c
int x;  // shared variable
```

Thread 1

Local Memory

Thread 2

Local Memory
Shared vs. Local Memory

```
int x; // shared variable
```

1. Read x
2. Store local copy of x
   Thread 1
   Local Memory

```
int x; // cached copy 1
```

3. Read x
4. Store local copy of x
   Thread 2
   Local Memory

```
int x; // cached copy 2
```
Shared vs. Local Memory

int x;  // shared variable

1. Write to local copy of x
Thread 1
Local Memory

int x;  // cached copy 1

2. Read from local copy of x
Thread 2
Local Memory

int x;  // cached copy 2
Shared vs. Local Memory

```
int x; // shared variable
```

1. Write to local copy of \( x \)
2. Update shared copy of \( x \)
3. Read \( x \)
4. Store local copy of \( x \)

```
int x; // cached copy 1
int x; // cached copy 2
```
Shared vs. Local Memory

How does one ensure that threads perceive the correct sequence of events?

• by means of synchronization events
• may imply mutual exclusion
• in Java, also imply thread cache reconciliation
Synchronization Events

Synchronization events are \textit{linearizable}:

- they are totally ordered
- all threads agree on the ordering
Synchronization Events

- **Explicit Lock**

```java
lock.lock();
try {
// critical section
}
finally {
lock.unlock();
}
```
Synchronization Events

• Using the synchronized keyword:

```java
public class ImportantObject {
    public synchronized void writeResult() {
        // critical section
    }
    public synchronized ResultObject readResult() {
        // critical section
    }
}
```
Synchronization Events

• **Using a synchronization block:**

```java
public class ImportantObject {
    public ResultObject readResult() {
        synchronized(ImportantObject.class) {
            // critical section
        }
    }
}
```
Volatile Fields

Use the volatile keyword when declaring a field.

- reads and writes effect memory reconciliation
- volatile fields are linearizable
- multiple reads-writes to volatile fields are not atomic
- mutual exclusion is still required
Volatile Fields

Use the volatile keyword when declaring a field.

- It is common to use volatile fields in cases where the fields are read by many threads but written to by only one thread.
public class ImportantObject
{
    private volatile int x;
    public int getValue()
    {
        return x;
    }
    public void writeValue(int value)
    {
        x = value;
    }
}
public class ImportantObject {
    private int x;
    public synchronized int getValue() {
        return x;
    }
    public synchronized void writeValue(int value) {
        x = value;
    }
}
Final Fields

A field declared using the final keyword cannot be modified once it has been initialized. Final fields are initialised in the object constructor.

But how does one ensure that multiple threads all see the same, initial value?
Final Fields

class FinalFieldExample
{
    final int x, int y;
    static FinalFieldExample f;
    public FinalFieldExample()
    {
        x = 3;
        y = 4;
    }
}

static void writer()
{
    f = new FinalFieldExample();
}

static void reader()
{
    if (f != null)
    {
        int i = f.x;
        int j = f.y;
    }
}
Final Fields

The single rule to remember is that the `this` keyword must not be released from the constructor before the constructor returns.
public class EventListener
{
    final int x;
    public EventListener(EventSource eventSource)
    {
        eventSource.registerListener(this);
    }
    public onEvent(Event e)
    {
        // handle the event
    }
}
Java memory Summary

Reads and writes are linearizable if:
• the field is volatile
• the field is protected by a unique lock that is acquired by all the readers and writers that access that field.