Desirable Properties of Concurrent Objects

**Safety Property**
- Nothing bad ever happens
- Needed for correctness

**Liveness Property**
- Something good eventually happens
- Needed for progress (e.g., no deadlock)
Correctness Properties

Sequential Consistency

- Method calls should appear to happen in a one-at-a-time sequential order
- For each thread method calls should appear to take effect in program order

Thread 1

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

Thread 2

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

Sequential Consistency

- Method calls should appear to happen in a one-at-a-time sequential order
- For each thread, method calls should appear to take effect in program order

Sequentially Consistent (one way)
Correctness Properties

Sequential Consistency

- Method calls should appear to happen in a one-at-a-time sequential order
- For each thread method calls should appear to take effect in program order

Thread 1

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

Thread 2

Sequentially Consistent (another way)
Correctness Properties

Sequential Consistency

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

— For each thread method calls should appear to take effect in program order

— Sequential Consistency is not compositional

Thread 1

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

Thread 2

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

Sequential Consistency

- Method calls should appear to happen in a one-at-a-time sequential order
- For each thread method calls should appear to take effect in program order
- Sequential Consistency is not compositional

$p$ and $q$ are independently sequentially consistent, but their composition is not
Correctness Properties

Linearizability

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

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

Thread 1

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

Thread 2

Time
Correctness Properties

Linearizability

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

 Thread 1

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

 Thread 2

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

Linearizable
Correctness Properties

Linearizability

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

— Compositional

Thread 1

Thread 2

Time
Correctness Properties

Linearizability

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

— Compositional

Thread 1

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

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

Thread 2

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

Not Linearizable
Correctness Properties

Linearizability

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

— Compositional

Thread 1

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

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

Thread 2

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

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

Linearizability

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

— Compositional

Thread 1

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

Thread 2

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

Linearizable (one way)
Correctness Properties

Linearizability

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

— Compositional

Thread 1

Thread 2

Linearizable
( another way )
Correctness Properties

Linearizability

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

— Compositional
Correctness Properties

Linearizability

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

— Compositional

Thread 1

push( x )  pop( y )  push( z )

Thread 2

push( y )  pop( x )

Not Linearizable
A Bounded Lock-Based Queue

```java
class BoundedQueue<T> {
    ReentrantLock enqLock, deqLock;
    Condition notEmptyCondition, notFullCondition;
    AtomicInteger size;
    Node head, tail;
    int capacity;
    public BoundedQueue(int _capacity) {
        capacity = _capacity;
        head = new Node(null);
        tail = head;
        size = new AtomicInteger(0);
        enqLock = new ReentrantLock();
        notFullCondition = enqLock.newCondition();
        deqLock = new ReentrantLock();
        notEmptyCondition = deqLock.newCondition();
    }
}
```

```java
protected class Node {
    public T value;
    public Node next;
    public Node(T x) {
        value = x;
        next = null;
    }
}
```

A Bounded Lock-Based Queue: Enqueue

```java
public void enq(T x) {
    boolean mustWakeDequeuers = false;
    enqLock.lock();
    try {
        while (size.get() == capacity)
            notFullCondition.await();
        Node e = new Node(x);
        tail.next = tail = e;
        if (size.getAndIncrement() == 0)
            mustWakeDequeuers = true;
    } finally {
        enqLock.unlock();
    }
    if (mustWakeDequeuers) {
        deqLock.lock();
        try {
            notEmptyCondition.signalAll();
        } finally {
            deqLock.unlock();
        }
    }
}
```

1. `public void enq(T x) {`
2.     `boolean mustWakeDequeuers = false;`
3.     `enqLock.lock( );`
4.     `try {
5.         `while (size.get() == capacity)
6.             `notFullCondition.await( );
7.         `Node e = new Node(x);
8.         `tail.next = tail = e;
9.         `if (size.getAndIncrement() == 0)
10.            `mustWakeDequeuers = true;
11.     } finally {
12.         `enqLock.unlock( );
13.     }
14.     `if (mustWakeDequeuers) {
15.         `deqLock.lock( );
16.         `try {
17.             `notEmptyCondition.signalAll( );
18.         } finally {
19.             `deqLock.unlock( );
20.         }
21.     }
22. }`
public T deq() {
    T result;
    boolean mustWakeEnqueuers = false;
    deqLock.lock();
    try {
        while (size.get() == 0)
            notEmptyCondition.await();
        result = head.next.value;
        head = head.next;
        if (size.getAndDecrement() == capacity)
            mustWakeEnqueuers = true;
    } finally {
        deqLock.unlock();
    }
    if (mustWakeEnqueuers)
        enqLock.lock();
    try {
        notFullCondition.signalAll();
    } finally {
        enqLock.unlock();
    }
    return result;
}
public T deq() {
    T result;
    boolean mustWakeEnqueuers = true;
    deqLock.lock();
    try {
        while (size.get() == 0)
           notEmptyCondition.await();
        result = head.next.value;
        head = head.next;
        if (size.getAndIncrement() == capacity) {
            mustWakeEnqueuers = true;
        }
    } finally {
        deqLock.unlock();
    }
    if (mustWakeEnqueuers) {
        enqLock.lock();
        try {
            notFullCondition.signalAll();
        } finally {
            enqLock.unlock();
        }
    }
    return result;
}
A Bounded Lock-Based Queue: Dequeue

```java
public T deq() {
    T result;
    boolean mustWakeEnqueuers = true;
    deqLock.lock();
    try {
        while (size.get() == 0)
            notEmptyCondition.await();
        result = head.next.value;
        head = head.next;
        if (size.getAndIncrement() == capacity)
            mustWakeEnqueuers = true;
    } finally {
        deqLock.unlock();
    }
    if (mustWakeEnqueuers)
        enqLock.lock();
        try {
            notFullCondition.signalAll();
        } finally {
            enqLock.unlock();
        }
    return result;
}
```

public T deq() {
    T result;
    boolean mustWakeEnqueuers = true;
    deqLock.lock();
    try {
        while (size.get() == 0)
            notEmptyCondition.await();
        result = head.next.value;
        head = head.next;
        if (size.getAndIncrement() == capacity)
            mustWakeEnqueuers = true;
    }
    finally {
        deqLock.unlock();
    }
    if (mustWakeEnqueuers) {
        enqLock.lock();
        try {
            notFullCondition.signalAll();
        }
        finally {
            enqLock.unlock();
        }
    }
    return result;
}
An Unbounded Lock-Free Queue

1. public class Node {
2.     public T value;
3.     public AtomicReference< Node > next;
4.     public Node(T _value) {
5.         value = _value;
6.         next = new AtomicReference< Node >(null);
7.     }
8. }

Source: Herlihy & Shavit.,
“The Art of Multiprocessor Programming”, 1st Edition
public void enq(T value) {
    Node node = new Node(value);
    while (true) {
        Node last = tail.get();
        Node next = last.next.get();
        if (last == tail.get()) {
            if (next == null) {
                if (last.next.compareAndSet(next, node)) {
                    tail.compareAndSet(last, node);
                    return;
                }
            }
            else { tail.compareAndSet(last, next); }
        }
    }
}
public T deq() throws EmptyException {
    while (true) {
        Node first = head.get();
        Node last = tail.get();
        Node next = first.next.get();
        if (first == head.get()) {
            if (first == last) {
                if (next == null) {
                    throw new EmptyException();
                }
            } else {
                tail.compareAndSet(last, next);
            }
        } else {
            T value = next.value;
            if (head.compareAndSet(first, next))
                return value;
        }
    }
}

Exponential Backoff

```java
public class Backoff {
  final int minDelay, maxDelay;
  int limit;
  final Random rand;

  public Backoff( int min, int max ) {
    minDelay = min;
    maxDelay = min;
    limit = minDelay;
    rand = new Random( );
  }

  public void backoff( ) throws InterruptedException {
    int delay = rand.nextInt( limit );
    limit = Math.min( maxDelay, 2 * limit );
    Thread.sleep( delay );
  }
}
```

**Source:** Herlihy & Shavit., “The Art of Multiprocessor Programming”, 1st Edition
An Unbounded Lock-Free Stack

1.  *public class* LockFreeStack< T > {
2.     AtomicReference< Node > top = new AtomicReference< Node >( null );
3.     *static final int* MIN_DELAY = ...;
4.     *static final int* MAX_DELAY = ...;
5.     Backoff backoff = new Backoff( MIN_DELAY, MAX_DELAY );

6.     *protected class* Node {
7.         *public* T value;
8.         *public* Node next;
9.         *public* Node( T _value ) {
10.             value = _value;
11.             next = null;
12.         }
13.     }

protected boolean tryPush(Node node) {
    Node oldTop = top.get();
    node.next = oldTop;
    return (top.compareAndSet(oldTop, node));
}

public void push(T value) {
    Node node = new Node(value);
    while (true) {
        if (tryPush(node)) { return; }
        else { backoff.backoff(); }
    }
}
An Unbounded Lock-Free Stack: Push

1. protected boolean tryPush( Node node ) {
2.     Node oldTop = top.get();
3.     node.next = oldTop;
4.     return ( top.compareAndSet( oldTop, node ) );
5. }
6. public void push( T value ) {
7.     Node node = new Node( value );
8.     while ( true ) {
9.         if ( tryPush( node ) ) { return; }
10.        else { backoff.backoff(); }  
11.    }
12. }

protected boolean tryPush( Node node ) {
    Node oldTop = top.get();
    node.next = oldTop;
    return (top.compareAndSet(oldTop, node));
}

public void push( T value ) {
    Node node = new Node( value );
    while (true) {
        if (tryPush(node)) { return; }
        else { backoff.backoff(); }
    }
}
An Unbounded Lock-Free Stack: Pop

protected Node tryPop() throws EmptyException {
    Node oldTop = top.get();
    if (oldTop == null) {
        throw new EmptyException();
    }
    Node newTop = oldTop.next;
    if (top.compareAndSet(oldTop, newTop)) {
        return oldTop;
    } else {
        return null;
    }
}

public T pop() throws EmptyException {
    while (true) {
        Node returnNode = tryPop();
        if (returnNode != null) {
            return returnNode.value;
        } else {
            backoff.backoff();
        }
    }
}

An Unbounded Lock-Free Stack: Pop

1. `protected Node tryPop( ) throws EmptyException {`
2. `Node oldTop = top.get( );`
3. `if ( oldTop == null ) {`
4. `throw new EmptyException( );`
5. `}`
6. `Node newTop = oldTop.next;`
7. `if ( top.compareAndSet( oldTop, newTop ) ) { return oldTop; }`
8. `else { return null; }`
9. `}`
10. `public T pop( ) throws EmptyException {`
11. `while ( true ) {`
12. `Node returnNode = tryPop( );`
13. `if ( returnNode != null ) { return returnNode.value; }
14. `else { backoff.backoff( ); }
15. `}
16. `}`

Elimination-Backoff Stack

A: return (b)
B: return ( )
C: return (d)

A: pop ( )
B: push (b)
C: pop ( )

Lock-Free Elimination Array

Shared Lock-Free Stack

top

d → e → f
Elimination Array

1. public class EliminationArray< T > {
2.    private static final int duration = ...;
3.    LockFreeExchanger< T >[ ] exchanger;
4.    Random rand;
5.    public EliminationArray( int capacity ) {
6.        exchanger = ( LockFreeExchanger< T >[ ] ) new LockFreeExchanger[ capacity ];
7.        for ( int i = 0; i < capacity; i++ ) {
8.            exchanger[ i ] = new LockFreeExchanger< T >();
9.        }
10.       rand = new Random( );
11.    }
12.    public T visit( T value, int range ) throws TimeoutException {
13.       int slot = rand.nextInt( range );
14.       return ( exchanger[ slot ].exchange( value, duration, TimeUnit.MILLISECONDS ) );
15.    }
16.  }

An Unbounded Lock-Free Elimination-Backoff Stack

```java
public class EliminationBackoffStack< T > extends LockFreeStack< T > {
    static final int capacity = ...;
    EliminationArray< T > eliminationArray = new EliminationArray< T >( capacity );
    static int range = ...;

    public void push( T value ) {
        Node node = new Node( value );
        while ( true ) {
            if ( tryPush( node ) ) { return; }
            else try {
                T otherValue = eliminationArray.visit( value, range );
                if ( otherValue == null ) { return; } // exchanged with pop
            } catch ( TimeoutException ex ) { }
        }
    }

```
public T pop() throws EmptyException {
    while (true) {
        Node returnNode = tryPop();
        if (returnNode != null) {
            return returnNode.value;
        } else try {
            T otherValue = eliminationArray.visit(null, range);
            if (otherValue != null) {
                return otherValue;
            }
        } catch (TimeoutException ex) { }
    }
}