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

\[ q.e\text{q}(x) \quad q.d\text{e}(y) \]

Thread 2

\[ q.e\text{q}(y) \quad q.d\text{e}(x) \]
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.

Thread 1

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

Thread 2

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

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

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
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

Thread 1

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

Thread 2

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

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

Thread 2
- `q.enq(y)`
- `q.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
q.enq(x) ← q.deq(y)
```

```
Thread 2
q.enq(y)
```
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

\[ q.enq(y) \]

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

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

Thread 2

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

Time
Correctness Properties

Linearizability

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

— Compositional

Thread 1

\[ q.enq(x) \]

\[ q.deq(y) \]

Thread 2

\[ q.enq(y) \]

\[ q.deq(x) \]

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

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

Thread 2

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

Linearizable (another way)
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)

Time
Correctness Properties

Linearizability

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

Not Linearizable
A Bounded Lock-Based Queue

1. \textbf{public class} BoundedQueue\textless T \textgreater{} {
2. \hspace{1em} \text{ReentrantLock} enqLock, deqLock;
3. \hspace{1em} \text{Condition} notEmptyCondition, notFullCondition;
4. \hspace{1em} \text{AtomicInteger} size;
5. \hspace{1em} \text{Node} head, tail;
6. \hspace{1em} \text{int} capacity;
7. \hspace{1em} \text{public} BoundedQueue( \text{int} \_capacity ) {
8. \hspace{2em} capacity = \_capacity;
9. \hspace{2em} head = \text{new Node( null );}
10. \hspace{2em} tail = head;
11. \hspace{2em} size = \text{new AtomicInteger( 0 );}
12. \hspace{2em} enqLock = \text{new ReentrantLock( );}
13. \hspace{2em} notFullCondition = enqLock.newCondition( );
14. \hspace{2em} deqLock = \text{new ReentrantLock( );}
15. \hspace{2em} notEmptyCondition = deqLock.newCondition( );
16. \hspace{1em} }

1. \textbf{protected class} Node {
2. \hspace{1em} \text{public} T value;
3. \hspace{1em} \text{public} Node next;
4. \hspace{1em} \text{public} Node( T x ) {
5. \hspace{2em} value = x;
6. \hspace{2em} next = null;
7. \hspace{2em} }
8. \hspace{1em} }

Source: Herlihy & Shavit., “The Art of Multiprocessor Programming”, 1\textsuperscript{st} Edition
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();
        }
    }
}
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();
        }
    }
}
```

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;
}
```


A Bounded Lock-Based Queue: Dequeue

1. `public T deq() {`
2. `T result;`
3. `boolean mustWakeEnqueuers = true;`
4. `deqLock.lock( );`
5. `try {`
6. `while ( size.get( ) == 0 )`
7. `notEmptyCondition.await( );`
8. `result = head.next.value;`
9. `head = head.next;`
10. `if ( size.getAndIncrement( ) == capacity ) {`
11. `mustWakeEnqueuers = true;`
12. `}`
13. `} finally { deqLock.unlock( ); }`
14. `if ( mustWakeEnqueuers ) {`
15. `enqLock.lock( );`
16. `try {`
17. `notFullCondition.signalAll( );`
18. `} finally { enqLock.unlock( ); }`
19. `}`
20. `return result;`
21. `}

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

1. `public class` Backoff {
2.  `final int` minDelay, maxDelay;
3.  `int` limit;
4.  `final` Random rand;
5.  `public` Backoff( `int` min, `int` max ) {
6.  `minDelay` = min;
7.  `maxDelay` = min;
8.  `limit` = `minDelay`;
9.  `rand` = `new` Random( );
10. }
11. `public void` backoff( ) `throws` InterruptedException {
12.  `int` delay = `rand`.nextInt( `limit` );
13.  `limit` = `Math.min`( `maxDelay`, `2 * limit` );
14.  `Thread.sleep`( `delay` );
15. }
16. }

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

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 ) ) {
8.       **return** oldTop;
9.   } else { **return** null; }
10. }
11. **public** T pop( ) **throws** EmptyException {
12.   while ( true ) {
13.       Node returnNode = tryPop( );
14.       **if** ( returnNode != null ) {
15.           **return** returnNode.value;
16.       } else { backoff.backoff( ); }
17.   }
18. }

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

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

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

Lock-Free Elimination Array

Shared Lock-Free Stack

top

D → E → F
public class EliminationArray< T > {

private static final int duration = ...;

LockFreeExchanger< T >[ ] exchanger;

Random rand;

public EliminationArray( int capacity ) {

    exchanger = ( LockFreeExchanger< T >[ ] ) new LockFreeExchanger[ capacity ];

    for ( int i = 0; i < capacity; i++ ) {

        exchanger[ i ] = new LockFreeExchanger< T >();

    }

    rand = new Random( );

}

public T visit( T value, int range ) throws TimeoutException {

    int slot = rand.nextInt( range );

    return ( exchanger[ slot ].exchange( value, duration, TimeUnit.MILLISECONDS ) );

}

}
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) { }
    }
}