Concurrent Objects

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit
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

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

\[ q \cdot \text{deq}() / \circ \]
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];
    }
}
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: $\text{head} = \text{tail}$
A Lock-Based Queue

```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];
    }
}
```

Initially `head = tail`
Lock-Based \( \text{deq}() \)
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();
    }
}
```

Acquire lock at method start
Check if Non-Empty

Not equal?

Waiting to enqueue…

Art of Multiprocessor Programming
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

Art of Multiprocessor Programming
Modify the Queue

Waiting to enqueue…
Implementation: `deq()`

```java
class Queue {
    private T[] items; // storage
    private int head; // next item to read
    private int tail; // next item to write

    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

1. Queue not empty?
2. Remove item and update head
3. Queue empty?
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

Art of Multiprocessor Programming
Release the Lock

head

0

1

y

tail

2

My turn!

x

6

7

5

4

3

Art of Multiprocessor Programming

22
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!
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();
    }
}
```

Should be correct because modifications are mutually exclusive...
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

head

0

1

x

y

5

4

3

2

tail

deq()

enq(z)

z

Art of Multiprocessor Programming
Wait-free 2-Thread Queue

head

result = x

queue[tail] = z

tail

Art of Multiprocessor Programming
Wait-free 2-Thread Queue

head++

head

0

1

tail

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 %
            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
- 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 \textit{state}
  – Usually given by a set of \textit{fields}
  – Queue example: sequence of items
• Each object has a set of \textit{methods}
  – Only way to manipulate state
  – Queue example: \texttt{enq} and \texttt{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
  – 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
Pre and PostConditions for Dequeue

- **Precondition:**
  - Queue is empty

- **Postcondition:**
  - Throws Empty exception

- **Postcondition:**
  - Queue state unchanged
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?
Methods Take Time
Methods Take Time

invocation 12:00

q.enq(○)

time

Art of Multiprocessor Programming

41
Methods Take Time

Method call

invocation 12:00

q.enq(0)
Methods Take Time

Invocation 12:00

q.enq(Ø)

Method call

time
Methods Take Time

Method call

Invocation 12:00

Response 12:01

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
Concurrent Methods Take Overlapping Time
Concurrent Methods Take Overlapping 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()` calls overlap?
    - Two `deq()` calls? `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

---

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
class Queue {
    private T[] items;
    private int head;
    private int tail;

    public T deque() 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.

Let's capture the idea of describing the concurrent via the sequential behavior.

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

time
Example

$q.enq(x)$

time
Example

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

\[ \text{time} \]
Example

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

time
Example

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

time

Art of Multiprocessor Programming
Example

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

q.enq(x)

time
Example

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

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

-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
Example

q.enq(x)

time
Example

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

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

`time`
Example

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

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

\[ \text{time} \]
Example

q.enq(x)

q.deq(x)

time

linearizable
Example

q.enq(x)

time
Example

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

```
q.enq(y)
```

```
time
```
Example

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

Art of Multiprocessor Programming
Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example
Example

Comme ci
Comme ça

multiple orders OK linearizable

g.enq(x)
g.enq(y)
g.deq(y)
g.deq(x)

time
Read/Write Register Example

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

time
Read/Write Register Example

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

write(1) already happened
Read/Write Register Example

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

write(1) already happened
Read/Write Register Example

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

write(1) already happened

Not linearizable.
Read/Write Register Example

write(0) → read(1) → write(2)

write(1) already happened

write(1)
Read/Write Register Example

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

write(1) already happened
Read/Write Register Example

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

write(1) already happened

not linearizable
Read/Write Register Example

write(0) -> write(1) -> write(2) -> read(1)

time
Read/Write Register Example

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

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

time
Read/Write Register Example

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

linearizable
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

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

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

thread
Invocation Notation

A q.enq(x)

thread  method
Invocation Notation

A q.enq(x)

thread

method

object
Invocation Notation

A q.enq(x)

thread method object arguments
Response Notation

A q: void
Response Notation
Response Notation

A \textit{q: void}

\texttt{thread} \hspace{1cm} \texttt{result}
Response Notation

A q: void

thread

object

result
Response Notation

Method is implicit

A q: void

thread

result

object
Response Notation

Method is implicit

thread

A q: empty()

exception

object
History - Describing an Execution

\[ H = \]

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

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

Sequence of invocations and responses
Definition

- Invocation & response *match* if

```
A q.enq(3)
A q:void
```
Object Projections

\[ H = \]

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

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

H|q =

B \texttt{q.deq()}
B \texttt{q: 3}
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

\[ H | B = \]

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

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

May or may not have taken effect
Complete Subhistory

\[ H = \]

\[ A \text{ q.enq}(3) \]
\[ A \text{ q:}void \]
\[ \text{box}(A \text{ q.enq}(5)) \]
\[ B \text{ p.enq}(4) \]
\[ B \text{ p:}void \]
\[ B \text{ q.deq}() \]
\[ B \text{ q:}3 \]

**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)
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
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)  match
A q: void  match
B p.enq(4)  match
B p: void  match
B q.deq()  Final pending
B q: 3  invocation OK
A q.enq(5)
Sequential Histories

Method calls do not interleave

match

match

match

Final pending invocation OK
Well-Formed Histories

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

Per-thread projections sequential

\[ H= \]

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

\[ H \mid B= \]

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

Per-thread projections sequential

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

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

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

Threads see the same thing in both

\[ H \mid A = G \mid A \]
\[ H \mid B = G \mid B \]
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 spec for $x$
Precedence

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

A method call \textit{precedes} another if response event \textit{precedes} invocation event
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 $\Rightarrow_G \subseteq \Rightarrow_S$
Remarks

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

• Condition $G \subseteq S$
  – Means that $S$ respects “real-time order” of $G$
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 $S$!
Example

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

Complete this pending invocation

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

Art of Multiprocessor Programming
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

time
Example

Equivalent sequential history

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
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

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

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

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

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

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();
    }
}
More Reasoning: Wait-free

```java
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;
    }
}
```
public class WaitFreeQueue {
    int head = 0, tail = 0;
    Item[] 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;
    }
}

Remember that there is only one enqueuer and only one dequeuer. Linearization order is order head and tail fields modified.
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
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 \( G \subseteq S \)

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
Example

q.enq(x)
Example

def q.enq(x):
    # Enqueue x

def q.deq(y):
    # Dequeue y

time
Example

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

- Blue arrows: Enqueue operations
- Red arrow: Dequeue operation

Timeline: Time progresses from left to right.
Example

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

- `q.enq(x)`: Insert `x` into the queue.
- `q.deq(y)`: Remove `y` from the queue.
- `q.enq(y)`: Insert `y` into the queue again.

Time progression:
- `time`
Example

not linearizable

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

\texttt{q.enq(x)}
\texttt{q.enq(y)}
\texttt{q.deq(y)}
\texttt{q.enq(x)}
\texttt{q.enq(y)}

Yet Sequential Consistent
Theorem

Sequential Consistency is not composable
FIFO Queue Example

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

.time
FIFO Queue Example

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

\text{time}
FIFO Queue Example

\[\text{History } H\]

\[\begin{align*}
\text{p.enq}(x) & \quad \text{q.enq}(x) & \quad \text{p.deq}(y) \\
\text{q.enq}(y) & \quad \text{p.enq}(y) & \quad \text{q.deq}(x)
\end{align*}\]
H|p Sequentially Consistent

\[
p.\text{enq}(x) \rightarrow q.\text{enq}(x) \rightarrow p.\text{deq}(y) \\
q.\text{enq}(y) \rightarrow p.\text{enq}(y) \rightarrow q.\text{deq}(x)
\]
H|q Sequentially Consistent

\[
\begin{align*}
&\text{p.enq}(x) & \text{q.enq}(x) & \text{p.deq}(y) \\
&\text{q.enq}(y) & \text{p.enq}(y) & \text{q.deq}(x)
\end{align*}
\]
Ordering imposed by p

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

time
Ordering imposed by q

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

time
Ordering imposed by both

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

Time
Combining orders

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

Timeline:
```
  time
```

Art of Multiprocessor Programming
Fact

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

- \texttt{x.write(1)}
- \texttt{y.write(1)}
- \texttt{y.read(0)}
- \texttt{x.read(0)}
The Flag Example

- Each thread’s view is sequentially consistent
  - It went first
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: 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!
Peterson's Algorithm

```java
public void lock() {
    flag[i] = true;
    victim = i;
    while (flag[j] && victim == i) {};
}
public void unlock() {
    flag[i] = false;
}
```
Crux of Peterson Proof

(1) \( \text{write}_B(\text{flag}[B]=\text{true}) \rightarrow \)

(3) \( \text{write}_B(\text{victim}=B) \rightarrow \)

(2) \( \text{write}_A(\text{victim}=A) \rightarrow \text{read}_A(\text{flag}[B]) \rightarrow \text{read}_A(\text{victim}) \)
Crux of Peterson Proof

1. $\text{write}_B(\text{flag}[B]=\text{true}) \Rightarrow$

2. $\text{write}_A(\text{victim}=A) \Rightarrow \text{read}_A(\text{flag}[B]) \Rightarrow \text{read}_A(\text{victim})$

3. $\text{write}_B(\text{victim}=B) \Rightarrow$

**Observation:** proof relied on fact that if a location is stored, a later load by some thread will return this or a later stored value.
Opinion: 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
Hardware Consistency

Initially, \( a = b = 0 \).

Processor 0

- `mov 1, a` ; Store
- `mov b, %ebx` ; Load

Processor 1

- `mov 1, b` ; Store
- `mov a, %eax` ; Load

What are the final possible values of \%eax and \%ebx after both processors have executed?

Sequential consistency implies that no execution ends with \%eax = \%ebx = 0

Slide used with permission of Charles E. Leiserson
Hardware Consistency

- No modern-day processor implements sequential consistency.
- Hardware actively reorders instructions.
- Compilers may reorder instructions, too.
- Why?
- Because most of performance is derived from a single thread’s unsynchronized execution of code.
Q. Why might the hardware or compiler decide to reorder these instructions?
A. To obtain higher performance by covering load latency — *instruction-level parallelism.*
Q. When is it safe for the hardware or compiler to perform this reordering?

A. When $a \neq b$.

A'. And there’s no concurrency.
Hardware Reordering

- Processor can issue stores faster than the network can handle them ⇒ store buffer.
- Loads take priority, bypassing the store buffer.
- Except if a load address matches an address in the store buffer, the store buffer returns the result.

Slide used with permission of Charles E. Leiserson
X86: Memory Consistency

Thread’s Code

- Store1
- Store2
- Load1
- Load2
- Store3
- Store4
- Load3
- Load4
- Load5

1. Loads are *not* reordered with loads.
2. Stores are *not* reordered with stores.
3. Stores are *not* reordered with prior loads.
4. A load *may be* reordered with a prior store to a different location *but not* with a prior store to the same location.
5. Stores to the same location respect a global total order.
X86: Memory Consistency

Thread’s Code

1. Loads are not reordered with loads.
2. Stores are not reordered with stores.
3. Stores are not reordered with prior loads.
4. A load may be reordered with a prior store to a different location but not with a prior store to the same location.
5. Stores to the same location respect a global total order.

Total Store Ordering (TSO)…weaker than sequential consistency

OK!
Memory Barriers (Fences)

- A *memory barrier* (or *memory fence*) is a hardware action that enforces an ordering constraint between the instructions before and after the fence.
- A memory barrier can be issued explicitly as an instruction (x86: mfence)
- The typical cost of a memory fence is comparable to that of an L2-cache access.
1. Loads are not reordered with loads.
2. Stores are not reordered with stores.
3. Stores are not reordered with prior loads.
4. A load may be reordered with a prior store to a different location but not with a prior store to the same location.
5. Stores to the same location respect a global total order.

Total Store Ordering + properly placed memory barriers = sequential consistency
Memory Barriers

• Explicit Synchronization
• Memory barrier will
  – Flush write buffer
  – Bring caches up to date
• Compilers often do this for you
  – Entering and leaving critical sections
Volatile Variables

• In Java, can ask compiler to keep a variable up-to-date by declaring it volatile

• Adds a memory barrier after each store

• Inhibits reordering, removing from loops, & other “compiler optimizations”

• Will talk about it in detail in later lectures
Summary: Real-World

- Hardware weaker than sequential consistency
- Can get sequential consistency at a price
- Linearizability better fit for high-level software
Linearizability

- Linearizability
  - Operation takes effect instantaneously between invocation and response
  - Uses sequential specification, locality implies composability
Summary: Correctness

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

- We will use *linearizability* as our consistency condition 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>Everyone makes progress</td>
<td><strong>Wait-free</strong></td>
<td><strong>Starvation-free</strong></td>
</tr>
<tr>
<td>Someone makes progress</td>
<td><strong>Lock-free</strong></td>
<td><strong>Deadlock-free</strong></td>
</tr>
</tbody>
</table>

Art of Multiprocessor Programming
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/.
- 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.