Concurrent Objects

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.enq( )
FIFO Queue: Dequeue Method

```python
q.deq() / Ø
```
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();
    }
}
```

Acquire lock at method start.
Check if Non-Empty

Waiting to enqueue…
Implementation: `deq()`

```java
public T deq() throws EmptyException {
    lock.lock();
    try {
        if (tail == head)
            throw new EmptyException();
        T x = items[head % items.length];
        head++;
        return x;
    } finally {
        lock.unlock();
    }
}
```

If queue empty throw exception
Modify the Queue

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

```java
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
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 the Lock

My turn!
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();
    }
}
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

deq()

head

0

1

x

y

tail

enq(z)

z

enq(z)

deq()
Wait-free 2-Thread Queue

result = x

head

queue[tail] = z

x

y

z
Wait-free 2-Thread Queue

head++

head

0

1

y

z

tail

tail--

x

6

7

5

4

3

2

0
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

Let's begin with correctness
Sequential Objects

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

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

- If (precondition)
  - the object is in such-and-such a state
  - before you call the method,
- Then (postcondition)
  - the method will return a particular value
  - or throw a particular exception.
- and (postcondition, con’t)
  - the object will be in some other state
  - 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(...)
Methods Take Time

invocation
12:00

q.enq(...)
Methods Take Time

```
invocation
12:00

q.enq(...)  Method call
```
Methods Take Time

invocation 12:00
g.enq(...)

response 12:01

void

Method call

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

Art of Multiprocessor Programming
Concurrent Methods Take Overlapping Time
Concurrent Methods Take Overlapping Time

Method call

Method call

time
Concurrent Methods Take Overlapping Time
Sequential vs Concurrent

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

• Concurrent
  – Because method calls overlap, object might \textit{never} be between method calls
Sequential vs Concurrent

• Sequential:
  – Each method described in isolation

• Concurrent
  – Must characterize *all* possible interactions with concurrent calls
    • What if two `enq`s overlap?
    • Two `deq`s? `enq` and `deq`? …
Sequential vs Concurrent

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

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

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

• Concurrent:
  – Everything can potentially interact with everything else

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”
Linearizability

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

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

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

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

$q.enq(x)$
Example

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

Art of Multiprocessor Programming
Example

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

q

Art of Multiprocessor Programming
Example

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

time
Example

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

Art of Multiprocessor Programming
Example
Example

q.enq(x)

time
Example

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

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

Art of Multiprocessor Programming
Example

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

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

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

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

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

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

not linearizable
Example
Example

\[ q . e n q ( x ) \]
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)

linearizable

time
Example

q.enq(x)

time
Example

\begin{align*}
q.\text{enq}(x) \\
q.\text{enq}(y)
\end{align*}
Example
Example

q.enq(x)
q.enq(y)
q.deq(y)
q.deq(x)
Comme ci
Comme ça

Example

multiple orders OK linearizable

Art of Multiprocessor Programming
Read/Write Register Example

![Diagram showing the sequence of read and write operations on a register. The sequence is as follows:
- write(0)
- read(1)
- write(2)
- write(1)
- read(0)

The diagram includes arrows indicating the flow of time from left to right.]
Read/Write Register Example

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

write(1) already happened
Read/Write Register Example

write(0) → read(1) → write(2) → write(1) already happened → read(0)
Read/Write Register Example

- write(0)
- read(1)
- write(2)
- write(1) already happened
- read(0)

(not linearizable)
Read/Write Register Example

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

write(1) already happened
Read/Write Register Example

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

write(1) already happened
Read/Write Register Example

write(0) → read(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
Read/Write Register Example

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

linearizable
Read/Write Register Example

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

time
Read/Write Register Example

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

time
Read/Write Register Example

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

time

Art of Multiprocessor Programming
Read/Write Register Example

Not 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 $q.\text{enq}(x)$
Invocation Notation

\texttt{A q.enq(x)}

thread
Invocation Notation

A\text{q.enq}(x)

thread \quad \text{method}
Invocation Notation

A thread

object

method

q.enq(x)
Invocation Notation

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

thread \quad \text{method} \quad \text{object} \quad \text{arguments}
Response Notation

A q: void
Response Notation

\[ \text{thread} \]

A q: void
Response Notation

A q: void

thread result

(2) Art of Multiprocessor Programming 105
Response Notation

A q: void

thread

object

result
Response Notation

Method is implicit

thread

object

A q: void

result
Response Notation

Method is implicit

A q: empty()
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

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

Thread names agree

Object names agree

Method call
Object Projections

\[ H = \]

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

\[ H|q = \]

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

\[ H|B = \]

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

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

\[ H = \]

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

May or may not have taken effect
Complete Subhistory

\[ H = \]

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

\[ \text{Complete}(H) = \]

A q.enq(3)
A q: void

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)

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
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
match
Sequential Histories

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

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

B q.deq()  match
B q:3

A q.enq(5)  Final pending invocation OK
Sequential Histories

A q.enq(3)
A q: void

B p.enq(4)
B p: void

B q.deq()
B q: 3

A q.enq(5)

Method calls of different threads do not interleave

match

match

Final pending invocation OK

match
Well-Formed Histories

\[ H = \]

A q.enq(3)
B p.enq(4)
B p:void
B q.deq()
A q:void
B q:3
Well-Formed Histories

Per-thread projections sequential

\[ H = \]

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

\[ H | B = \]

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

\[ H | A = \]

A \ q.\ enq(3)  
A \ q:\ void
Equivalent Histories

Threads see the same thing in both

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

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

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

A method call precedes another if response event precedes invocation event

Method call

Art of Multiprocessor Programming
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 \( m_0 \xrightarrow{H} m_1 \), if
  – \( m_0 \) precedes \( m_1 \)

• Relation \( m_0 \xrightarrow{H} m_1 \) 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 $\rightarrow_G \subset \rightarrow_S$

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

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

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

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

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

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

Complete this pending invocation
Example

B.q.enq(4)
B q: void
B q: deq()
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: 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: 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)
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
Concurrenty

- **Question:** When does linearizability require a method invocation to block?
- **Answer:** never.
- **Linearizability is** *non-blocking*
Non-Blocking Theorem

If method invocation
   \texttt{A q.inv(...)}

is pending in history \( H \), then there exists a response
   \texttt{A q:res(...)}

such that
   \( H + A q:res(...) \)

is linearizable
Proof

• Pick linearization $S$ of $H$
• If $S$ already contains
  – Invocation $A \text{ q.inv(...)}$ and response,
  – Then we are done.
• Otherwise, pick a response such that
  – $S + A \text{ q.inv(...)} + A \text{ q:res(...)}$
  – 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;
    Object[] 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;
    }
}

Linearization order is order head and tail fields modified

Remember that there is only one enqueuer and only one dequeuer
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 $S$
  
\[ \text{Where } G \subset S \]

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

(5) Art of Multiprocessor Programming 160
Example

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

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

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

\[ \text{time} \]
Example

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

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

(5)
Example

`q.enq(x)`

`q.enq(y)`

`q.deq(y)`

`q.enq(x)`

`q.enq(y)`

not linearizable

q.er q(y)

q.er q(x)

q.deq(y)

time

(5)
Example

Yet Sequentially Consistent

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

(5)

Art of Multiprocessor Programming
Theorem

Sequential Consistency is not composable
FIFO Queue Example

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

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

time
FIFO Queue Example

History H
H|p Sequentially Consistent

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

Time
H|q Sequentially Consistent

\[
p.\text{enq}(x) \rightarrow q.\text{enq}(x) \rightarrow p.\text{deq}(y) \rightarrow q.\text{enq}(y) \rightarrow p.\text{enq}(y) \rightarrow q.\text{deq}(x)
\]
Ordering imposed by p

Art of Multiprocessor Programming
Ordering imposed by q

$p$.enqueue($x$) $\rightarrow$ $q$.enqueue($x$) $\rightarrow$ $p$.dequeue($y$) $\rightarrow$ $q$.enqueue($y$) $\rightarrow$ $p$.enqueue($y$) $\rightarrow$ $q$.dequeue($x$)

time
Ordering imposed by both

Art of Multiprocessor Programming
Combining orders

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

Art of Multiprocessor Programming
176
Fact

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

\begin{itemize}
\item x.write(1)
\item y.write(1)
\item y.read(0)
\item x.read(0)
\end{itemize}
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 1: It’s Wrong

- This pattern
  - Write mine, read yours
- Is exactly the flag principle
  - Beloved of Alice and Bob
  - Heart of mutual exclusion
    - Peterson
    - Bakery, etc.
- It’s non-negotiable!
Opinion 2: But It Feels So Right …

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

<table>
<thead>
<tr>
<th></th>
<th>Non-Blocking</th>
<th>Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.